



TigerDB RDBMS Concepts

Anil Jagtap

TigerDB Database Server

A Practical RDBMS Guide with TLang and TigerDB Router



For developers, administrators, application teams, and business users

Version basis: This edition documents the protected stable TigerDB Server line, the TLang Phase 4.4 frontend language, and the Router R8 operational line. Future experimental branches should be tested against these baselines before promotion.

Contents

Part I: RDBMS Foundations

- Why Databases Exist
- The Relational Model
- Tables, Columns, and Records
- Keys, Relationships, and Integrity
- Indexes and Query Performance
- Logs, Durability, and Recovery
- Users, Roles, and Secure Access
- Designing Small Databases Well

Part II: TigerDB Database Server

- Introducing TigerDB
- Installing, Building, and Starting TigerDB
- Connecting, Logging In, and Changing Passwords
- Databases, Tables, and Storage Layout
- Data Definition Commands
- Data Modification Commands
- SELECT Queries and Result JSON
- Joins, Grouping, Ordering, and Limits
- Indexes and Query Planning
- Functions and Analytics
- Stored Procedures in TigerDB
- Stored Procedure Control Flow
- Security, Licensing, and Administration
- Performance and Benchmarking
- TigerConnect Drivers and Client Tools
- TigerDB Command Reference

Part III: TLang Frontend Programming Language

- Introducing TLang
- TLang Program Structure
- Variables, Expressions, and Control Flow
- Screens, Forms, Menus, and Tables
- Connecting TLang to TigerDB
- RecordSets and FOREACH
- Calling Stored Procedures from TLang
- A Complete TLang Application

Part IV: TigerDB Router

- Router Architecture
- Router Configuration
- Routing Databases to Backends
- Router Users, Grants, and Stored Procedure Policy
- Router Health, Clients, and Reload
- Router Security and Operations

Part V: Appendices

- TigerDB Server Configuration Examples
- Router Configuration Examples
- Command Cookbook
- Troubleshooting Checklist
- Glossary

Part I

RDBMS Foundations

1. Why Databases Exist

A database exists so that valuable information can be stored, found, changed, protected, and recovered in a predictable way. A relational database adds a disciplined structure to this work: named databases, tables, columns, keys, queries, roles, and logs.

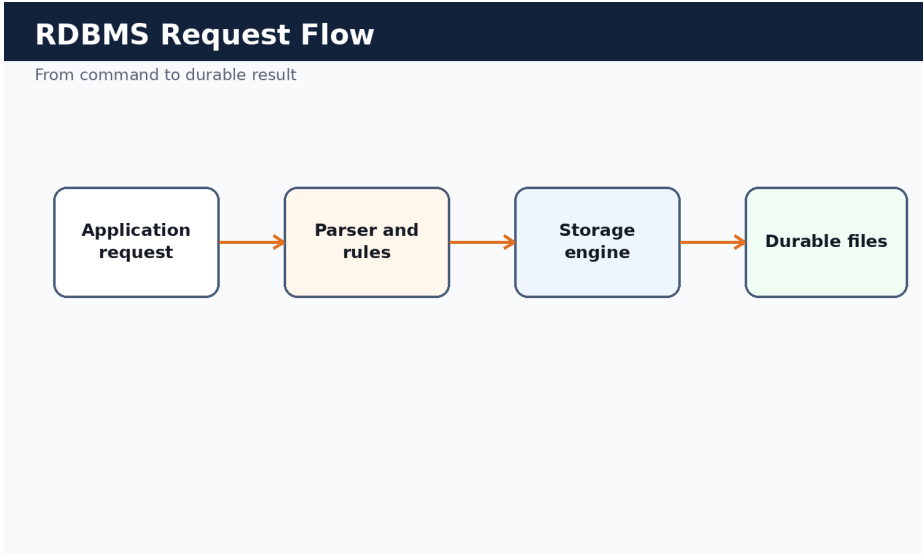


Figure. Why Databases Exist in relational database work.

Concept

Why Databases Exist is one of the ideas that separates a database from a simple collection of files. A file can contain values, but a database gives those values a name, a shape, permissions, and predictable behavior after restart.

The relational model is powerful because it turns information into tables of rows and columns. Once data has a stable shape, applications can ask clear questions and administrators can reason about security, performance, and recovery.

The practical value is repeatability. A well-defined command should mean the same thing tomorrow as it does today. That is why relational systems place so much emphasis on keys, constraints, indexes, logs, and controlled user access.

TigerDB follows these same ideas. It exposes commands through a JSON-framed protocol, stores data in per-database structures, records changes through write-ahead logging, and uses users and roles to determine what an authenticated caller can do.

Why it matters

A business application rarely fails because it stores too little data. It fails when the data loses meaning: duplicate customers, missing keys, inconsistent reports, or users who can modify records they should only read.

A strong RDBMS design gives every important object a purpose. The database name separates business domains, the table name describes a set of facts, the primary key

identifies one record, and an index makes a repeated search fast enough to use in real applications.

Good design also keeps operational work boring. Backups, logs, checkpoint files, and configuration settings are not visible to most users, but they decide whether a restart is routine or stressful.

Practical example

```
CREATE DATABASE sales;
USE sales;
CREATE TABLE customers (id INT PRIMARY KEY, name STRING, status STRING);
INSERT INTO customers VALUES (1, 'Anil', 'active');
SELECT * FROM customers;
```

Common mistakes

- Treating a database as only a flat file with a nicer interface.
- Creating tables before deciding what uniquely identifies each row.
- Giving every application administrator-level credentials.
- Ignoring recovery until after a failure occurs.

Checklist

- Give every table a clear purpose and a clear primary key.
- Use smaller examples to test the meaning of a command before applying it to production data.
- Prefer named roles and repeatable procedures over ad hoc manual changes.
- Record configuration choices so performance and security decisions can be reproduced.

2. The Relational Model

The relational model organizes data into relations, commonly represented as tables. Each row represents one fact or entity instance. Each column represents a named attribute. Relationships appear through matching key values rather than through hidden links.

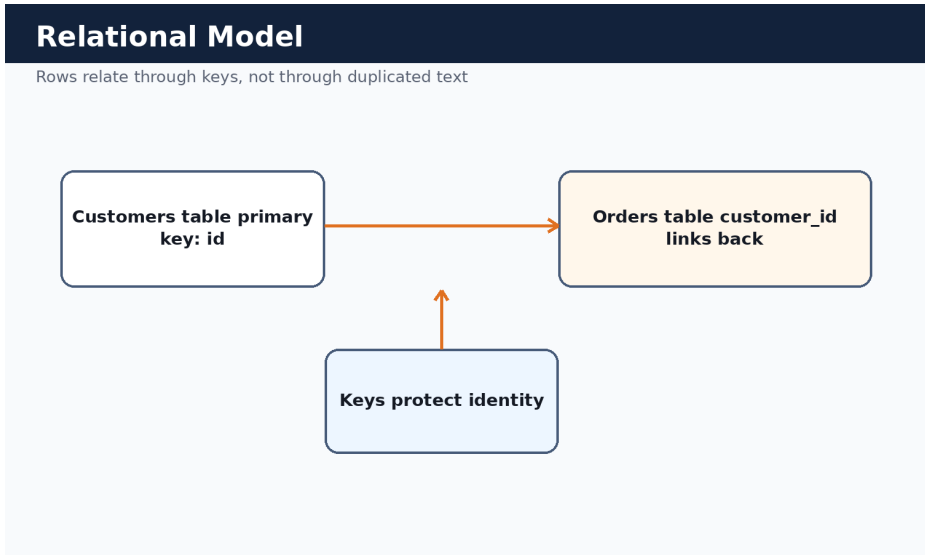


Figure. The Relational Model in relational database work.

Concept

The Relational Model is one of the ideas that separates a database from a simple collection of files. A file can contain values, but a database gives those values a name, a shape, permissions, and predictable behavior after restart.

The relational model is powerful because it turns information into tables of rows and columns. Once data has a stable shape, applications can ask clear questions and administrators can reason about security, performance, and recovery.

The practical value is repeatability. A well-defined command should mean the same thing tomorrow as it does today. That is why relational systems place so much emphasis on keys, constraints, indexes, logs, and controlled user access.

TigerDB follows these same ideas. It exposes commands through a JSON-framed protocol, stores data in per-database structures, records changes through write-ahead logging, and uses users and roles to determine what an authenticated caller can do.

Why it matters

A business application rarely fails because it stores too little data. It fails when the data loses meaning: duplicate customers, missing keys, inconsistent reports, or users who can modify records they should only read.

A strong RDBMS design gives every important object a purpose. The database name separates business domains, the table name describes a set of facts, the primary key

identifies one record, and an index makes a repeated search fast enough to use in real applications.

Good design also keeps operational work boring. Backups, logs, checkpoint files, and configuration settings are not visible to most users, but they decide whether a restart is routine or stressful.

Practical example

```
CREATE TABLE customers (id INT PRIMARY KEY, name STRING);
CREATE TABLE orders (id INT PRIMARY KEY, customer_id INT, amount DOUBLE);
INSERT INTO customers VALUES (101, 'North Shop');
INSERT INTO orders VALUES (1, 101, 250.75);
SELECT customers.name, orders.amount
FROM customers JOIN orders ON customers.id = orders.customer_id;
```

Common mistakes

- Copying the customer name into every order instead of using a key.
- Using vague column names that do not tell the reader what the value means.
- Assuming the application alone can protect consistency.
- Designing reports before the relationships are clear.

Checklist

- Give every table a clear purpose and a clear primary key.
- Use smaller examples to test the meaning of a command before applying it to production data.
- Prefer named roles and repeatable procedures over ad hoc manual changes.
- Record configuration choices so performance and security decisions can be reproduced.

3. Tables, Columns, and Records

A table is the main unit of organization in a relational database. A column describes one property, a row contains one record, and the schema is the contract that tells the database and application what values are expected.

Concept

Tables, Columns, and Records is one of the ideas that separates a database from a simple collection of files. A file can contain values, but a database gives those values a name, a shape, permissions, and predictable behavior after restart.

The relational model is powerful because it turns information into tables of rows and columns. Once data has a stable shape, applications can ask clear questions and administrators can reason about security, performance, and recovery.

The practical value is repeatability. A well-defined command should mean the same thing tomorrow as it does today. That is why relational systems place so much emphasis on keys, constraints, indexes, logs, and controlled user access.

TigerDB follows these same ideas. It exposes commands through a JSON-framed protocol, stores data in per-database structures, records changes through write-ahead logging, and uses users and roles to determine what an authenticated caller can do.

Why it matters

A business application rarely fails because it stores too little data. It fails when the data loses meaning: duplicate customers, missing keys, inconsistent reports, or users who can modify records they should only read.

A strong RDBMS design gives every important object a purpose. The database name separates business domains, the table name describes a set of facts, the primary key identifies one record, and an index makes a repeated search fast enough to use in real applications.

Good design also keeps operational work boring. Backups, logs, checkpoint files, and configuration settings are not visible to most users, but they decide whether a restart is routine or stressful.

Practical example

```
CREATE TABLE products (
  id INT PRIMARY KEY,
  sku STRING,
  name STRING,
  price DOUBLE,
  active BOOL
);
INSERT INTO products VALUES (1, 'SKU-100', 'Sample Product', 19.95, true);
SELECT id, sku, price FROM products WHERE active = true;
```

Common mistakes

- Using one wide table for unrelated subjects.

- Changing column meaning without changing the name.
- Storing booleans as arbitrary strings such as yes, Y, true, and 1 in the same column.
- Skipping sample data tests before application integration.

Checklist

- Give every table a clear purpose and a clear primary key.
- Use smaller examples to test the meaning of a command before applying it to production data.
- Prefer named roles and repeatable procedures over ad hoc manual changes.
- Record configuration choices so performance and security decisions can be reproduced.

4. Keys, Relationships, and Integrity

Keys give identity to rows. Primary keys identify records. Foreign-key-like relationships connect records across tables. Even when a system does not enforce every relationship automatically, the application and procedure design should treat keys as the foundation of integrity.

Concept

Keys, Relationships, and Integrity is one of the ideas that separates a database from a simple collection of files. A file can contain values, but a database gives those values a name, a shape, permissions, and predictable behavior after restart.

The relational model is powerful because it turns information into tables of rows and columns. Once data has a stable shape, applications can ask clear questions and administrators can reason about security, performance, and recovery.

The practical value is repeatability. A well-defined command should mean the same thing tomorrow as it does today. That is why relational systems place so much emphasis on keys, constraints, indexes, logs, and controlled user access.

TigerDB follows these same ideas. It exposes commands through a JSON-framed protocol, stores data in per-database structures, records changes through write-ahead logging, and uses users and roles to determine what an authenticated caller can do.

Why it matters

A business application rarely fails because it stores too little data. It fails when the data loses meaning: duplicate customers, missing keys, inconsistent reports, or users who can modify records they should only read.

A strong RDBMS design gives every important object a purpose. The database name separates business domains, the table name describes a set of facts, the primary key identifies one record, and an index makes a repeated search fast enough to use in real applications.

Good design also keeps operational work boring. Backups, logs, checkpoint files, and configuration settings are not visible to most users, but they decide whether a restart is routine or stressful.

Practical example

```
CREATE TABLE stores (id INT PRIMARY KEY, name STRING);
CREATE TABLE sales (id INT PRIMARY KEY, store_id INT, amount DOUBLE);
INSERT INTO stores VALUES (10, 'Central Store');
INSERT INTO sales VALUES (1, 10, 750.25);
SELECT sales.id, stores.name, sales.amount
FROM sales JOIN stores ON sales.store_id = stores.id;
```

Common mistakes

- Using names as primary keys when names can change.
- Leaving records without a meaningful identifier.
- Relying on row position instead of a key.

- Creating relationships that are only understood by one developer.

Checklist

- Give every table a clear purpose and a clear primary key.
- Use smaller examples to test the meaning of a command before applying it to production data.
- Prefer named roles and repeatable procedures over ad hoc manual changes.
- Record configuration choices so performance and security decisions can be reproduced.

5. Indexes and Query Performance

Indexes are auxiliary structures that help the database find rows faster. A primary-key lookup should be quick, and a secondary index can make repeated searches practical. Indexes have a cost: they must be maintained when data is inserted, updated, or deleted.

Index Lookup

Indexes reduce search work and improve repeated lookups

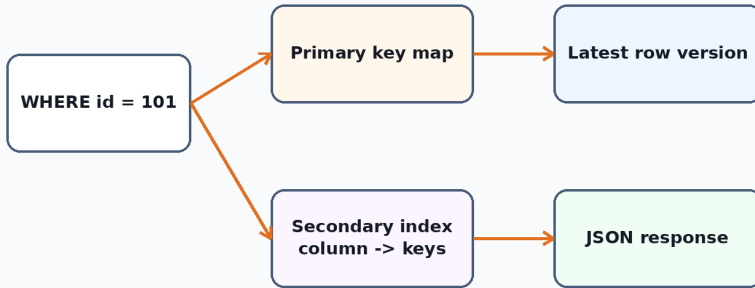


Figure. Indexes and Query Performance in relational database work.

Concept

Indexes and Query Performance is one of the ideas that separates a database from a simple collection of files. A file can contain values, but a database gives those values a name, a shape, permissions, and predictable behavior after restart.

The relational model is powerful because it turns information into tables of rows and columns. Once data has a stable shape, applications can ask clear questions and administrators can reason about security, performance, and recovery.

The practical value is repeatability. A well-defined command should mean the same thing tomorrow as it does today. That is why relational systems place so much emphasis on keys, constraints, indexes, logs, and controlled user access.

TigerDB follows these same ideas. It exposes commands through a JSON-framed protocol, stores data in per-database structures, records changes through write-ahead logging, and uses users and roles to determine what an authenticated caller can do.

Why it matters

A business application rarely fails because it stores too little data. It fails when the data loses meaning: duplicate customers, missing keys, inconsistent reports, or users who can modify records they should only read.

A strong RDBMS design gives every important object a purpose. The database name separates business domains, the table name describes a set of facts, the primary key

identifies one record, and an index makes a repeated search fast enough to use in real applications.

Good design also keeps operational work boring. Backups, logs, checkpoint files, and configuration settings are not visible to most users, but they decide whether a restart is routine or stressful.

Practical example

```
CREATE TABLE customers (id INT PRIMARY KEY, name STRING, city STRING);
CREATE INDEX idx_customers_city ON customers(city);
SELECT * FROM customers WHERE city = 'Mumbai';
SHOW INDEXES ON customers;
```

Common mistakes

- Creating an index for every column without measuring write overhead.
- Expecting an index to help a query that does not filter or order by the indexed value.
- Forgetting that indexed inserts do more work than unindexed inserts.
- Benchmarking only small tables.

Checklist

- Give every table a clear purpose and a clear primary key.
- Use smaller examples to test the meaning of a command before applying it to production data.
- Prefer named roles and repeatable procedures over ad hoc manual changes.
- Record configuration choices so performance and security decisions can be reproduced.

6. Logs, Durability, and Recovery

Durability means committed work should survive a restart. Write-ahead logging records changes before they become part of normal table state. Checkpoints and recovery logic keep restart time manageable while preserving correctness.

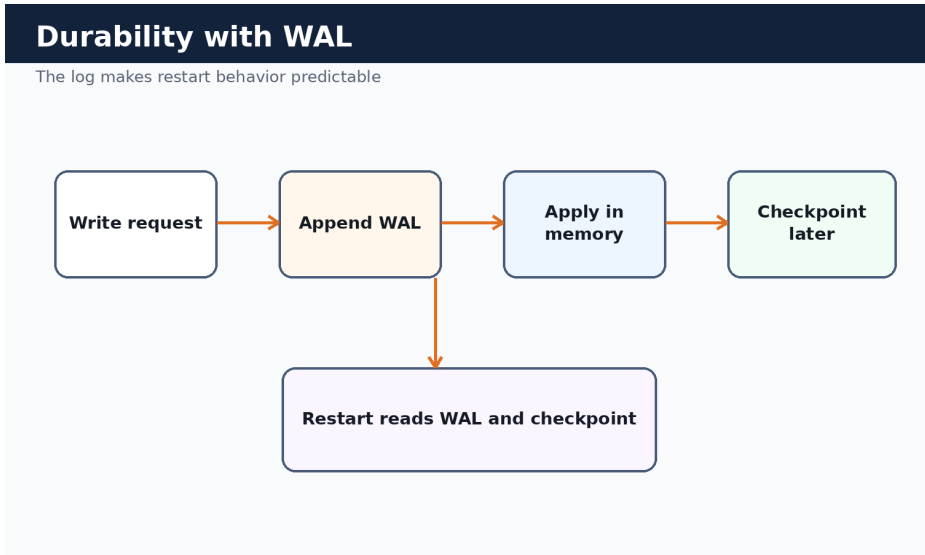


Figure. Logs, Durability, and Recovery in relational database work.

Concept

Logs, Durability, and Recovery is one of the ideas that separates a database from a simple collection of files. A file can contain values, but a database gives those values a name, a shape, permissions, and predictable behavior after restart.

The relational model is powerful because it turns information into tables of rows and columns. Once data has a stable shape, applications can ask clear questions and administrators can reason about security, performance, and recovery.

The practical value is repeatability. A well-defined command should mean the same thing tomorrow as it does today. That is why relational systems place so much emphasis on keys, constraints, indexes, logs, and controlled user access.

TigerDB follows these same ideas. It exposes commands through a JSON-framed protocol, stores data in per-database structures, records changes through write-ahead logging, and uses users and roles to determine what an authenticated caller can do.

Why it matters

A business application rarely fails because it stores too little data. It fails when the data loses meaning: duplicate customers, missing keys, inconsistent reports, or users who can modify records they should only read.

A strong RDBMS design gives every important object a purpose. The database name separates business domains, the table name describes a set of facts, the primary key

identifies one record, and an index makes a repeated search fast enough to use in real applications.

Good design also keeps operational work boring. Backups, logs, checkpoint files, and configuration settings are not visible to most users, but they decide whether a restart is routine or stressful.

Practical example

```
INSERT INTO sales VALUES (1, 101, 750.25);  
SHOW WAL;  
SHOW STATS;  
SHOW PERFORMANCE;
```

Common mistakes

- Judging a database only by in-memory speed.
- Disabling durability without a clear reason.
- Ignoring WAL growth and checkpoint behavior.
- Running recovery tests only after production is live.

Checklist

- Give every table a clear purpose and a clear primary key.
- Use smaller examples to test the meaning of a command before applying it to production data.
- Prefer named roles and repeatable procedures over ad hoc manual changes.
- Record configuration choices so performance and security decisions can be reproduced.

7. Users, Roles, and Secure Access

Secure access begins with identity. A database user logs in, receives permissions, and performs only the actions allowed by the system. Roles keep these decisions understandable and auditable.

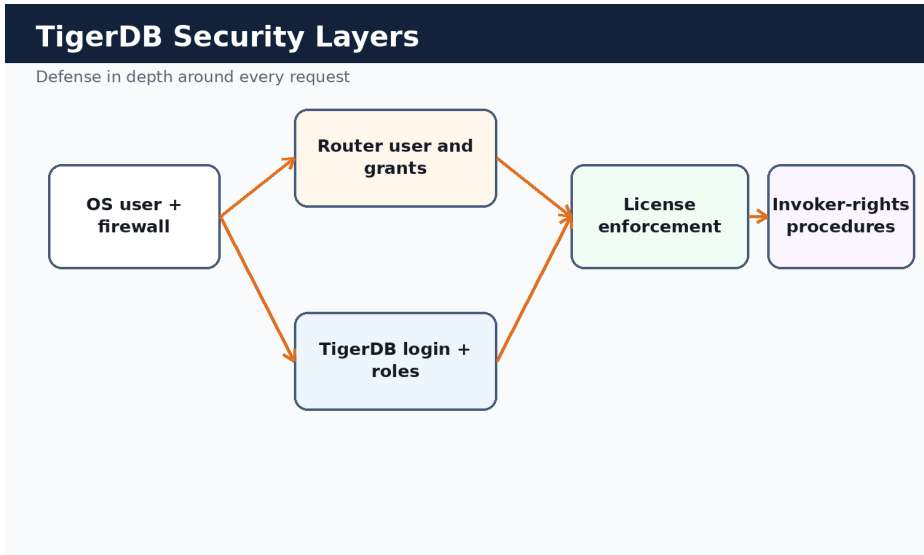


Figure. Users, Roles, and Secure Access in relational database work.

Concept

Users, Roles, and Secure Access is one of the ideas that separates a database from a simple collection of files. A file can contain values, but a database gives those values a name, a shape, permissions, and predictable behavior after restart.

The relational model is powerful because it turns information into tables of rows and columns. Once data has a stable shape, applications can ask clear questions and administrators can reason about security, performance, and recovery.

The practical value is repeatability. A well-defined command should mean the same thing tomorrow as it does today. That is why relational systems place so much emphasis on keys, constraints, indexes, logs, and controlled user access.

TigerDB follows these same ideas. It exposes commands through a JSON-framed protocol, stores data in per-database structures, records changes through write-ahead logging, and uses users and roles to determine what an authenticated caller can do.

Why it matters

A business application rarely fails because it stores too little data. It fails when the data loses meaning: duplicate customers, missing keys, inconsistent reports, or users who can modify records they should only read.

A strong RDBMS design gives every important object a purpose. The database name separates business domains, the table name describes a set of facts, the primary key

identifies one record, and an index makes a repeated search fast enough to use in real applications.

Good design also keeps operational work boring. Backups, logs, checkpoint files, and configuration settings are not visible to most users, but they decide whether a restart is routine or stressful.

Practical example

```
LOGIN admin IDENTIFIED BY 'admin123';  
CHANGE PASSWORD TO 'new_admin_password';  
CREATE USER report_user IDENTIFIED BY 'report123';  
GRANT READ ON sales TO report_user;
```

Common mistakes

- Sharing the same account across unrelated applications.
- Using administrator credentials for reporting jobs.
- Forgetting forced password change for first-time users.
- Leaving backend ports open when Router is intended to be the entry point.

Checklist

- Give every table a clear purpose and a clear primary key.
- Use smaller examples to test the meaning of a command before applying it to production data.
- Prefer named roles and repeatable procedures over ad hoc manual changes.
- Record configuration choices so performance and security decisions can be reproduced.

8. Designing Small Databases Well

Small databases teach the same lessons as large ones. A compact schema with clear keys, tested commands, and predictable security rules is easier to scale than a large design built without discipline.

Concept

Designing Small Databases Well is one of the ideas that separates a database from a simple collection of files. A file can contain values, but a database gives those values a name, a shape, permissions, and predictable behavior after restart.

The relational model is powerful because it turns information into tables of rows and columns. Once data has a stable shape, applications can ask clear questions and administrators can reason about security, performance, and recovery.

The practical value is repeatability. A well-defined command should mean the same thing tomorrow as it does today. That is why relational systems place so much emphasis on keys, constraints, indexes, logs, and controlled user access.

TigerDB follows these same ideas. It exposes commands through a JSON-framed protocol, stores data in per-database structures, records changes through write-ahead logging, and uses users and roles to determine what an authenticated caller can do.

Why it matters

A business application rarely fails because it stores too little data. It fails when the data loses meaning: duplicate customers, missing keys, inconsistent reports, or users who can modify records they should only read.

A strong RDBMS design gives every important object a purpose. The database name separates business domains, the table name describes a set of facts, the primary key identifies one record, and an index makes a repeated search fast enough to use in real applications.

Good design also keeps operational work boring. Backups, logs, checkpoint files, and configuration settings are not visible to most users, but they decide whether a restart is routine or stressful.

Practical example

```
CREATE DATABASE training;
USE training;
CREATE TABLE lessons (id INT PRIMARY KEY, title STRING, complete BOOL);
INSERT INTO lessons VALUES (1, 'Tables and keys', false);
UPDATE lessons SET complete = true WHERE id = 1;
SELECT * FROM lessons;
```

Common mistakes

- Skipping the small model because the final system will be bigger.
- Designing tables around screens instead of facts.
- Creating unclear naming conventions.

- Assuming performance problems can be fixed after every application query is written.

Checklist

- Give every table a clear purpose and a clear primary key.
- Use smaller examples to test the meaning of a command before applying it to production data.
- Prefer named roles and repeatable procedures over ad hoc manual changes.
- Record configuration choices so performance and security decisions can be reproduced.

Part II

TigerDB Database Server

9. Introducing TigerDB

TigerDB is a practical relational database server with multiple physical databases, per-database storage, write-ahead logging, users and roles, indexes, JSON responses, stored procedures, licensing, and performance-oriented read/write paths.

TigerDB is designed around a small set of direct ideas: a client connects, authenticates, selects a database, sends SQL or JSON actions, and receives JSON responses. This simple contract makes it usable from command-line tools, drivers, TLang programs, and Router-managed applications.

The stable server line documents in this book is the R4c performance baseline. It preserves the stored-procedure system, large SQL batch support, stable write and read paths, and a safer ORDER BY improvement. Future experimental branches should be compared against this baseline rather than replacing it casually.

TigerDB is not presented as a clone of any single commercial database. It is a focused database engine with a native JSON protocol, a lightweight stored-procedure model, and a Router that can centralize application access to multiple backend servers.

The project also includes TigerConnect drivers for Python, Java, C, and Go. Each driver uses one language package with two connection modes: direct database server mode and Router mode. This keeps the application API consistent while allowing deployment flexibility.

TigerDB Server Architecture

Stable server line with licensing, WAL, procedures, and performance fast paths

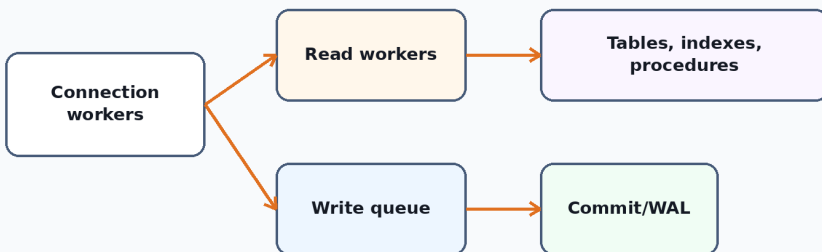


Figure. TigerDB Server combines workers, WAL, tables, indexes, procedures, and licensing into one server process.

Stable source rule: The R4c stable source should be protected. New work should be copied into experimental files, benchmarked, and promoted only after functionality, security, and performance remain intact.

10. Installing, Building, and Starting TigerDB

TigerDB is built as a C server. The normal optimized build links pthread support and the math library. The database uses local configuration and data directories, and the first successful setup should verify licensing, login, and basic SQL before application work begins.

Before running production data, create a clean directory layout and decide whether the server will be reached directly or through TigerDB Router. Direct server access is suitable for local development and simple deployments. Router access is preferable when applications need one gateway to multiple backend TigerDB servers.

After the server starts, the first checks should be simple: PING, LOGIN, SHOW LICENSE, SHOW DATABASES, CREATE DATABASE, USE, CREATE TABLE, INSERT, and SELECT. A small end-to-end test catches misconfiguration faster than a large application test.

If the server is protected by a firewall, verify that only intended clients can reach port 9191. When Router is used, backend port 9191 should usually be private and Router port 9292 should be the application-facing endpoint.

First smoke-test commands

```
PING;  
LOGIN admin IDENTIFIED BY 'admin123';  
SHOW LICENSE;  
SHOW DATABASES;
```

Build hygiene: Avoid running benchmarks against a debug or sanitizer build unless the purpose is diagnostic. Use an optimized build when recording performance numbers.

11. Connecting, Logging In, and Changing Passwords

TigerDB supports login, forced password change for first-time users, and password changes later when a user requests them. Clients can use SQL LOGIN syntax or JSON actions, depending on the driver or tool.

The authentication flow is intentionally explicit. A client connects to the TCP port, sends a login request, and receives a JSON response. If the response says a password must be changed, the client should prompt for a new password before sending normal database commands.

Changing the default administrative password is a basic security step. A first-time user should not continue to use the temporary password beyond initial setup. The same habit should apply to Router credentials and to any backend credentials stored in Router configuration.

Password change is not allowed inside TigerDB stored procedures. A procedure should contain business data logic, not session or credential management. This separation keeps authentication behavior clear and prevents procedure bodies from changing a caller's security state.

When applications connect through Router, the Router authenticates its own users and then logs into the backend TigerDB server using the configured backend credentials. Password synchronization improvements ensure a password change can update the Router-side path when appropriate.

SQL login and password change

```
LOGIN admin IDENTIFIED BY 'admin123';
CHANGE PASSWORD TO 'new_admin_password';
LOGIN admin IDENTIFIED BY 'new_admin_password';
```

Equivalent JSON actions

```
{"action": "login", "username": "admin", "password": "admin123"}
{"action": "change_password", "new_password": "new_admin_password"}
```

12. Databases, Tables, and Storage Layout

TigerDB supports multiple physical databases. Each database has its own storage area, catalog, tables, WAL, checkpoint files, and procedure directory. This physical separation makes the deployment model easier to understand.

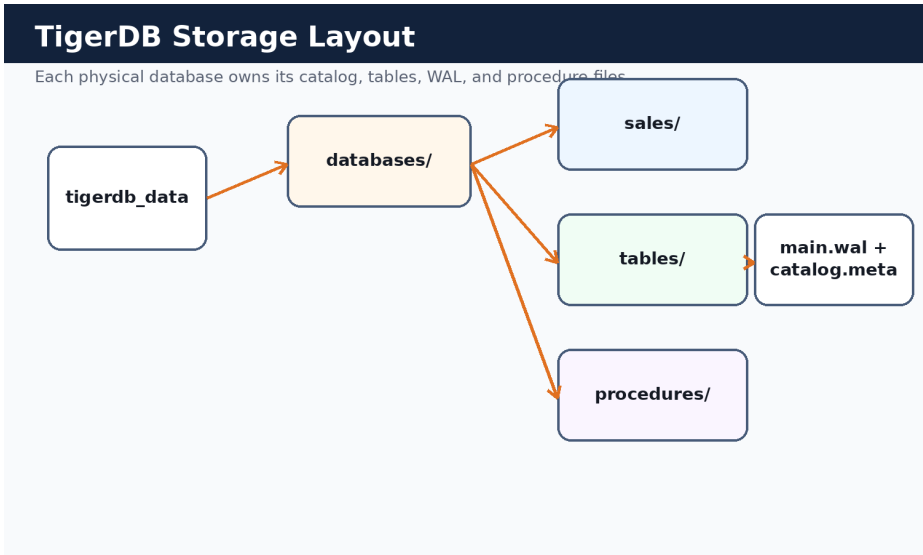


Figure. TigerDB stores each database under its own directory with tables, WAL, catalog, checkpoint, and procedures.

The default database is available for simple use, but production applications should generally create named databases that match business domains such as sales, inventory, billing, or analytics. Named databases make permissions and Router routes easier to maintain.

A table lives inside a database. TigerDB internally resolves the visible table name to a database-qualified internal name so that tables in different databases can have the same visible name without colliding.

The `config_db` system database stores internal administrative data such as users, user roles, and stored-procedure metadata. Application data should remain in application databases, not in `config_db`.

Procedure files are stored separately under the database procedure directory. Their metadata, including path and hash, is stored in the internal catalog. This is a key difference between TigerDB procedures and traditional fully in-catalog procedures.

Working with database names

```

CREATE DATABASE sales;
CREATE DATABASE inventory;
SHOW DATABASES;
USE sales;
SHOW TABLES;
  
```

13. Data Definition Commands

Data definition commands create the structures that hold data. In TigerDB, the main data definition commands are CREATE DATABASE, USE, CREATE TABLE, CREATE INDEX, CREATE UNIQUE INDEX, and stored-procedure definition commands.

A CREATE TABLE command should be treated as a contract between the database and the application. Column names should be clear, data types should match the business meaning, and the primary key should be chosen before large amounts of data are inserted.

TigerDB column types include integer, double, string, boolean, and timestamp-oriented values. The examples in this book use INT, DOUBLE, STRING, BOOL, and TIMESTAMP names in a simple way so that new readers can focus on structure before advanced reporting.

Indexes are also data definition objects. A secondary index on a frequently searched column can improve read performance, but it adds cost to writes. Benchmarking both indexed and unindexed inserts is useful when designing ingest-heavy applications.

Procedure definition commands are covered in depth later, but they belong in the DDL family because they create, show, and drop reusable named database logic.

Creating a table and indexes

```
CREATE TABLE customers (  
  id INT PRIMARY KEY,  
  name STRING,  
  city STRING,  
  amount DOUBLE,  
  active BOOL  
);  
  
CREATE INDEX idx_customers_city ON customers(city);  
CREATE UNIQUE INDEX idx_customers_name ON customers(name);  
SHOW TABLE customers;  
SHOW INDEXES ON customers;
```

Command	Purpose	Typical permission
CREATE DATABASE name	Create a physical database	FULL
USE name	Select current database	READ/WRITE/FULL on database
CREATE TABLE ...	Create table schema	FULL
CREATE INDEX ...	Create secondary index	FULL
SHOW TABLES	List tables	READ
SHOW TABLE table	Describe one table	READ

14. Data Modification Commands

Data modification commands change stored records. TigerDB supports INSERT, multi-row INSERT, UPDATE, and DELETE. Every write should be understood as both a table change and a WAL-recorded operation.

INSERT adds new records. The performance-optimized server line supports large multi-row insert statements, which lets a client send many rows in one request. Larger batches reduce protocol, parser, and dispatch overhead.

UPDATE changes existing rows that match a condition. UPDATE should be written carefully because a broad condition can modify many rows. A practical habit is to run a SELECT with the same WHERE clause before executing a sensitive UPDATE.

DELETE removes records that match a condition. As with UPDATE, DELETE is safer when the WHERE clause is tested first. In business applications, it is often better to mark a row inactive than to delete it immediately.

Every modifying command still runs through authorization checks. A user with READ should not be able to INSERT, UPDATE, or DELETE. Stored procedures also execute with invoker rights, so they do not bypass these rules.

Inserting, updating, and deleting

```
INSERT INTO customers VALUES (1, 'Anil', 'Pune', 250.75, true);
```

```
INSERT INTO customers VALUES
(2, 'Neha', 'Mumbai', 500.00, true),
(3, 'Kiran', 'Delhi', 125.50, false);
```

```
UPDATE customers SET amount = 600.00 WHERE id = 2;
DELETE FROM customers WHERE id = 3;
```

Batch size: Large INSERT batches are useful for loading data. Keep each request below the protocol message limit and measure batch sizes such as 5,000, 10,000, and 20,000 rows for your deployment.

15. SELECT Queries and Result JSON

SELECT is the main read command. TigerDB returns native JSON responses, making it easy for drivers, TLang programs, Flask clients, and scripts to consume results without a separate result-set protocol.

A SELECT can return all columns or a projection. Projection is usually better for application code because it moves only the values the application actually needs. SELECT * is useful for exploration, tests, and full-scan benchmarks.

TigerDB JSON responses typically include an ok field and result information such as rows. Client libraries can wrap this structure into row lists, RecordSets, or language-specific result objects.

WHERE clauses should be designed around indexed keys when performance matters. A primary-key lookup is very different from a full scan. The benchmark scripts in the project measure full scans, primary-key lookups, secondary-index lookups, and ORDER BY paths separately.

Because responses are JSON, application code should not parse printed tables. It should use the structured fields returned by the server or driver wrapper.

Basic SELECT forms

```
SELECT * FROM customers;
SELECT id, name, amount FROM customers WHERE id = 1;
SELECT id, city FROM customers WHERE city = 'Mumbai';
```

Simplified JSON result shape

```
{
  "ok": true,
  "rows": [
    {"id":1,"name":"Anil","amount":250.75}
  ]
}
```

16. Joins, Grouping, Ordering, and Limits

TigerDB supports practical query features such as JOIN, LEFT OUTER JOIN, GROUP BY, ORDER BY, and LIMIT. These features turn tables into reports and application-ready result sets.

JOIN combines related rows from two tables. A typical join uses a primary key from one table and a related value from another. LEFT OUTER JOIN keeps rows from the left table even when the right side does not match.

GROUP BY summarizes rows. It is useful for totals, counts, and averages by category. Grouped queries should be written with clear aggregate functions so that the meaning of each output column is obvious.

ORDER BY sorts the result. The stable R4c server line includes a safer ORDER BY improvement while preserving high-speed writes and full scans. Sorting is still more expensive than scanning, so applications should use it when the ordering matters.

LIMIT is a practical reporting tool. It reduces response size and helps screens and APIs avoid moving more rows than the user can inspect.

Reports with JOIN, GROUP BY, ORDER BY, and LIMIT

```
SELECT customers.name, sales.amount
FROM customers JOIN sales ON customers.id = sales.customer_id;

SELECT city, COUNT(*), SUM(amount), AVG(amount)
FROM customers
GROUP BY city;

SELECT id, name, amount FROM customers
ORDER BY amount DESC
LIMIT 10;
```

Stable Performance Profile

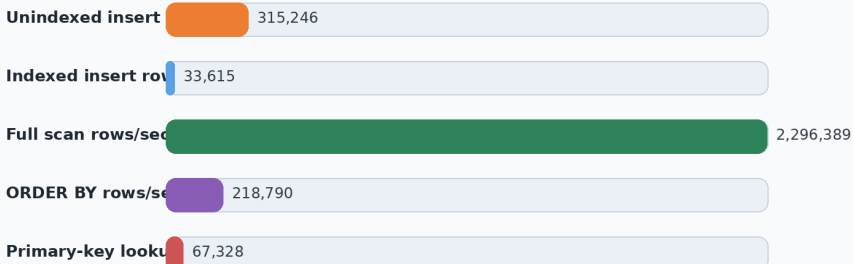


Figure. Example stable performance profile shows why query shape and indexing matter.

17. Indexes and Query Planning

Indexes are a central performance tool in TigerDB. The primary-key map supports direct identity lookup, while secondary indexes help repeated searches by another column.

The main tradeoff is simple: indexes make selected reads faster and writes more expensive. Every inserted or updated row may require index maintenance. For a reporting-heavy application, that cost is often worthwhile. For a high-speed ingest pipeline, it must be measured.

TigerDB benchmarks showed strong full-scan and primary-key lookup performance in the stable R4c line. Secondary-index reads can return large result sets, so the returned row volume matters as much as the query count.

An index is most useful when it reduces the number of rows that must be inspected. An index on a low-cardinality column such as active may return too many rows to help much. An index on a frequently searched identifier or category may be more valuable.

Use SHOW INDEXES to inspect which indexes exist and keep naming conventions consistent. Names such as idx_customers_city or idx_sales_customer_id are easier to understand than generic names such as idx1.

Index design example

```
CREATE INDEX idx_sales_customer_id ON sales(customer_id);
CREATE INDEX idx_sales_month ON sales(month);
SHOW INDEXES ON sales;
SELECT * FROM sales WHERE customer_id = 101;
```

Index choice	Good for	Cost
Primary key	Single-row identity lookup	Required unique key maintenance
Secondary index	Repeated WHERE on one column	Additional write maintenance
Unique secondary index	Enforcing uniqueness-like access	Higher validation cost
No index	Bulk ingest and full scans	Slower selective search

18. Functions and Analytics

TigerDB includes scalar, aggregate, statistical, and time-series-style functions. These functions let reports and analytics use the database as a calculation engine instead of pushing every calculation into application code.

Scalar functions operate on one value at a time. Examples include ABS, ROUND, SQRT, LOG, LOG10, EXP, POW, FLOOR, CEIL, CEILING, and SIGN. These are useful in calculated columns and reports.

Aggregate functions summarize a set of rows. COUNT, SUM, AVG, MEAN, MIN, MAX, MED, MEDIAN, VAR, STDDEV, STDDEV_POP, and STDDEV_SAMP support common reporting and statistical work.

Time-series and financial-style functions such as SMA, EMA, WMA, ROC, MOMENTUM, LOGRET, VOLATILITY, and RSI should be used with ORDER BY because their meaning depends on row order.

Function queries should be tested with small known data before being used in production reports. It is easier to verify a standard deviation, moving average, or RSI calculation when the input rows are visible.

Function examples

```
SELECT ROUND(amount), FLOOR(amount), CEIL(amount)
FROM sales;

SELECT city, COUNT(*), AVG(amount), STDDEV_POP(amount)
FROM customers
GROUP BY city;

SELECT ts, close,
       SMA(close, 20), EMA(close, 20), WMA(close, 20), RSI(close, 14)
FROM ticks
ORDER BY ts;
```

Category	Functions
Scalar	ABS, ROUND, SQRT, LOG, LN, LOG10, EXP, POW, FLOOR, CEIL, CEILING, SIGN
Aggregate	COUNT, SUM, AVG, MEAN, MIN, MAX, MED, MEDIAN, VAR, STDDEV, STDDEV_POP, STDDEV_SAMP
Window/time-series	SMA, EMA, WMA, ROC, MOMENTUM, LOGRET, VOLATILITY, RSI

19. Stored Procedures in TigerDB

TigerDB stored procedures are named .sp scripts registered in the database with trusted internal metadata. They are different from traditional database procedures because the body is external while the catalog, path, owner, hash, and enabled status are controlled internally.

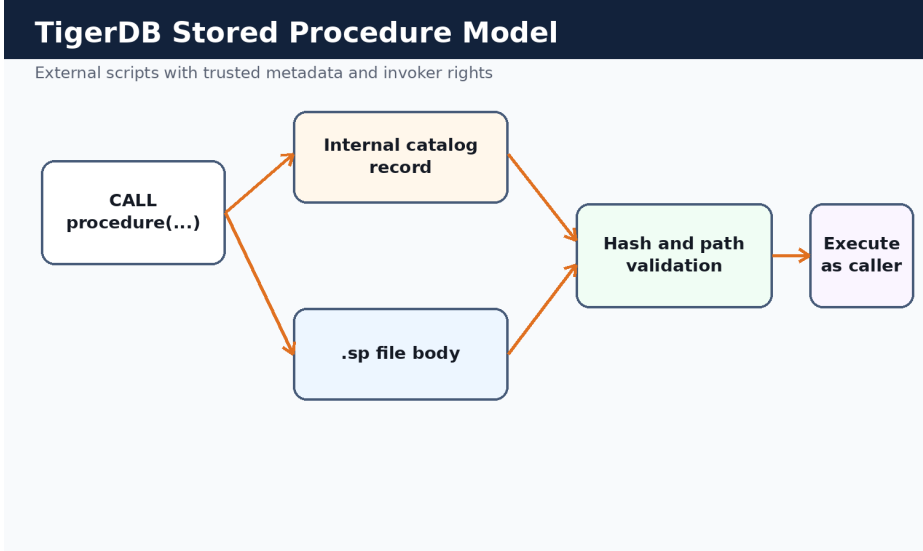


Figure. TigerDB procedures combine external .sp files with internal catalog metadata and hash validation.

The design is intentionally transparent. The procedure body is a script file, but TigerDB does not blindly trust it. Before execution, TigerDB checks the internal record, the canonical path, the file type, the file size, and the SHA-style hash. If the file was tampered with, execution is denied.

Procedures run with invoker rights. The user who calls the procedure must have the permission required by the SQL statements inside it. A procedure is not a back door to creator privileges.

Commands that change sessions, credentials, database context, or procedure definitions are deliberately forbidden inside procedures. CALL, LOGIN, USE, CHANGE PASSWORD, CREATE PROCEDURE, and DROP PROCEDURE stay outside procedure bodies.

This approach makes procedures useful for reusable business logic while keeping the database engine smaller and easier to reason about.

Basic stored procedure lifecycle

```
CREATE PROCEDURE list_customers
ON customers
AS $$
SELECT id, name, amount FROM customers
$$;

SHOW PROCEDURES;
SHOW PROCEDURES ON customers;
CALL list_customers();
DROP PROCEDURE list_customers ON customers;
```

Invoker rights: If the caller cannot run a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` directly, the caller should not gain that ability through a procedure call.

20. Stored Procedure Parameters and Variables

TigerDB procedures support IN parameters, local variables, SET assignment, and multi-statement procedure bodies. Parameters are passed by the caller; variables are internal temporary values used while the procedure runs.

Parameter declarations use a simple IN name TYPE pattern. The procedure body uses placeholders such as :p_id and :p_amount. The CALL statement supplies positional arguments, and TigerDB validates the values against the declared parameter types.

Local variables are declared with DECLARE. They can have default values and can be used as placeholders in SQL statements. SET changes local variables, but input parameters are read-only.

Multi-statement procedures run statements in order. Each executable SQL statement still goes through the normal TigerDB execution path and permission checks. The result wrapper can include internal steps and the last SQL result.

The supported data types for parameters and variables are intentionally small and practical: INT, DOUBLE, STRING, and BOOL. This keeps the procedure language useful without making it overly complex.

Parameters and variables

```
CREATE PROCEDURE record_sale
ON sales
(IN p_id INT, IN p_customer_id INT, IN p_amount DOUBLE)
AS $$
DECLARE v_bucket STRING DEFAULT 'small';

INSERT INTO sales(id, customer_id, amount, bucket)
VALUES (:p_id, :p_customer_id, :p_amount, :v_bucket);

SELECT id, customer_id, amount, bucket FROM sales WHERE id = :p_id
$$;

CALL record_sale(1, 101, 750.25);
```

Type	Example	Use
INT	101	Identifiers, counters
DOUBLE	750.25	Amounts, measurements
STRING	'active'	Names, categories, labels
BOOL	true	Flags and decisions

21. Stored Procedure Control Flow

TigerDB stored procedures support IF, ELSEIF, ELSE, ENDIF and WHILE, ENDWHILE. The control-flow language is intentionally limited, making it suitable for business rules without becoming a full general-purpose scripting engine.

IF/ELSEIF/ELSE/ENDIF lets a procedure select one branch based on values passed in or variables calculated in the procedure. Conditions support comparison operators, logical AND/OR/NOT, and parentheses.

WHILE/ENDWHILE repeats a group of statements while a condition remains true. The loop feature is useful for controlled generation or limited repeated work. Every loop should move toward completion through a SET assignment.

Loop safety limits protect the server from accidental infinite procedure execution. Nesting and execution-step limits are part of the procedure model, and they are documented so users can design procedures that remain predictable.

Only WHILE is included at this stage. FOR loops, cursors, recursive CALL, and procedure DDL inside procedures are deliberately out of scope.

IF/ELSEIF/ELSE/ENDIF

```
CREATE PROCEDURE classify_sale
ON sales
(IN p_id INT, IN p_amount DOUBLE)
AS $$
DECLARE v_bucket STRING DEFAULT 'small';

IF :p_amount >= 1000 THEN
SET v_bucket = 'large';
ELSEIF :p_amount >= 500 THEN
SET v_bucket = 'medium';
ELSE
SET v_bucket = 'small';
ENDIF;

UPDATE sales SET bucket = :v_bucket WHERE id = :p_id;
SELECT id, amount, bucket FROM sales WHERE id = :p_id
$$;
```

WHILE/ENDWHILE

```
CREATE PROCEDURE seed_numbers
ON numbers
(IN p_limit INT)
AS $$
DECLARE v_i INT DEFAULT 1;

WHILE :v_i <= :p_limit DO
INSERT INTO numbers(id, label) VALUES (:v_i, 'generated');
SET v_i = :v_i + 1;
ENDWHILE;

SELECT id, label FROM numbers
$$;
```

22. Security, Licensing, and Administration

TigerDB combines database users, roles, password changes, mandatory licensing, and operational diagnostics. These pieces protect data access and ensure that one-license-per-instance activation is respected.

Licensing is mandatory in the current model. Diagnostic actions such as PING and SHOW LICENSE can be allowed for troubleshooting, but normal database actions require a valid license state.

The license configuration includes a license key, license server host and port, timeout values, check interval, and local cache path. Local files such as server.id and tigerdb.license belong under the TigerDB data directory.

Users and roles determine database access. Administrators should create separate users for applications, reporting, ingest, and administration instead of sharing one account.

Security is strongest when combined with operating-system permissions and firewall rules. Run the server as a dedicated OS user where practical and expose only the network ports needed by the deployment model.

TigerDB license configuration

```
LICENSE_KEY=your-license-key
LICENSE_SERVER_HOST=159.223.89.223
LICENSE_SERVER_PORT=9443
LICENSE_SERVER_TIMEOUT_SECONDS=10
LICENSE_CHECK_INTERVAL_SECONDS=3600
LICENSE_CACHE_FILE=./tigerdb_data/license/tigerdb.license
```

Administrative diagnostics

```
SHOW LICENSE;
SHOW STATS;
SHOW MEMORY;
SHOW WAL;
SHOW WORKERS;
SHOW PERFORMANCE;
```

23. Performance and Benchmarking

Performance work should be measured, not guessed. TigerDB includes scripts and stable benchmark methods for inserts, indexed inserts, full scans, primary-key lookups, secondary-index lookups, and ORDER BY.

The stable R4c line restored high-speed writes and full scans while keeping a safer ORDER BY improvement. The latest large-scale test showed strong unindexed insert, excellent full-scan speed, healthy primary-key lookup speed, and a known bottleneck around large indexed insert workloads.

Insert throughput improves when rows are batched because the server pays fewer request, parser, and dispatch costs. Good batch sizes depend on row width and the protocol message limit. Batches of 5,000 to 20,000 rows were useful in the measured setup.

Indexed inserts are expected to be slower than unindexed inserts because index structures must be updated. If indexed ingest becomes a bottleneck, optimize secondary-index maintenance in a separate experimental branch, not by changing the stable baseline.

Read tests should separate full scans, primary-key lookups, secondary-index lookups, and ORDER BY. A single averaged read number hides which path is actually slow.

Benchmark matrix

```
python3 benchmark_insert.py --rows 500000 --batch-size 20000
python3 benchmark_insert.py --rows 500000 --batch-size 20000 --indexed
python3 benchmark_select.py --rows 500000 --mode fullscan --runs 3
python3 benchmark_select.py --rows 500000 --mode order-by --runs 3
python3 benchmark_select.py --rows 500000 --mode primary-key --lookups 10000
python3 benchmark_select.py --rows 500000 --mode secondary-index --lookups 2000
```

Stable Performance Profile

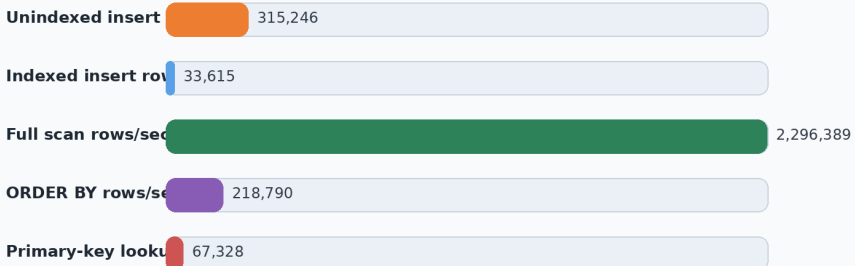


Figure. Representative R4c benchmark profile from the stable line.

24. TigerConnect Drivers and Client Tools

TigerConnect drivers give Python, Java, C, and Go applications a consistent API. Each language uses one driver with two modes: direct database mode and Router mode.

Driver Modes

One driver per language, two connection modes

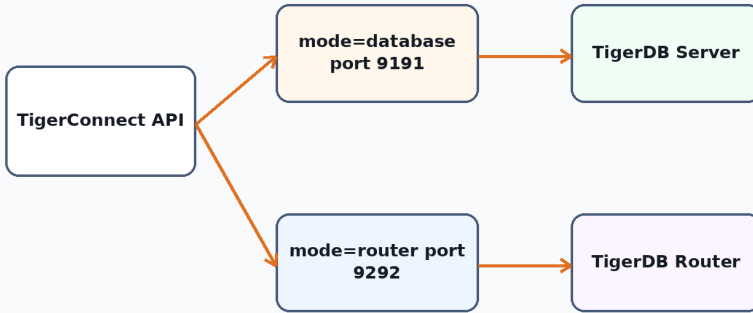


Figure. TigerConnect uses one driver per language with database and Router connection modes.

The direct database mode connects to port 9191 by default. Router mode connects to port 9292 by default. The main application operations remain similar: connect, login, use database, execute, query, call procedures, and close.

The Python driver also supports a legacy object-created-first style so older examples can remain readable. Java, C, and Go drivers include similar simple patterns for creating a connection object and using it repeatedly.

A simple C command-line client is useful for direct server testing. It can connect, prompt for credentials, detect first-time password changes, run SQL commands, and change passwords later when requested.

Applications should prefer driver methods and structured JSON results instead of parsing text. Stored procedure calls can be treated as SQL CALL commands or as driver helper methods.

Python legacy-style example

```

from tigerdb_connect import tigerdb_connect

db = tigerdb_connect('127.0.0.1', 9191)
db.login('admin', 'admin1234')
db.use_database('sales')
rows = db.query('SELECT * FROM customers;')
print(rows)
db.close()
  
```

Go Router-mode example

```
db, err := tigerconnect.Connect(tigerconnect.Config{
    Host: "127.0.0.1",
    Port: 9292,
    Mode: tigerconnect.ModeRouter,
    Username: "admin",
    Password: "admin1234",
    Database: "sales",
})
```

25. TigerDB Command Reference

This chapter collects the practical TigerDB Server command set in one place. The command forms are intentionally compact so the reader can use the chapter as a reference while developing.

Command	Example	Purpose
PING	PING;	Protocol and server reachability check
LOGIN	LOGIN admin IDENTIFIED BY 'pass';	Authenticate a user
CHANGE PASSWORD	CHANGE PASSWORD TO 'newpass';	Change current user password
SHOW LICENSE	SHOW LICENSE;	Inspect license state
SHOW STATS	SHOW STATS;	Inspect request and write counters
SHOW PERFORMANCE	SHOW PERFORMANCE;	Inspect performance-oriented diagnostics
SHOW MEMORY	SHOW MEMORY;	Inspect memory information
SHOW WAL	SHOW WAL;	Inspect WAL information
SHOW WORKERS	SHOW WORKERS;	Inspect worker configuration

Command	Example	Purpose
CREATE DATABASE	CREATE DATABASE sales;	Create a physical database
SHOW DATABASES	SHOW DATABASES;	List databases
USE	USE sales;	Select current database
CREATE TABLE	CREATE TABLE customers (...);	Create a table
SHOW TABLES	SHOW TABLES;	List tables in current database
SHOW TABLE	SHOW TABLE customers;	Describe a table
INSERT	INSERT INTO customers VALUES (...);	Add rows
UPDATE	UPDATE customers SET amount=10 WHERE id=1;	Modify rows
DELETE	DELETE FROM customers WHERE id=1;	Delete rows

Command	Example	Purpose
SELECT	SELECT * FROM customers;	Read rows
JOIN	SELECT ... FROM a JOIN b ON ...;	Combine matching rows
LEFT OUTER JOIN	SELECT ... FROM a LEFT OUTER JOIN b ON ...;	Keep left rows without matches

GROUP BY	SELECT city, COUNT(*) FROM customers GROUP BY city;	Aggregate by category
ORDER BY	SELECT * FROM sales ORDER BY amount DESC;	Sort results
LIMIT	SELECT * FROM sales LIMIT 10;	Limit rows returned

Command	Example	Purpose
CREATE INDEX	CREATE INDEX idx_city ON customers(city);	Create secondary index
CREATE UNIQUE INDEX	CREATE UNIQUE INDEX idx_name ON customers(name);	Create unique-style index
SHOW INDEXES	SHOW INDEXES ON customers;	List indexes
CREATE PROCEDURE	CREATE PROCEDURE p ON t AS \$ \$... \$\$;	Register procedure
SHOW PROCEDURES	SHOW PROCEDURES ON t;	List procedures
DROP PROCEDURE	DROP PROCEDURE p ON t;	Remove procedure
CALL	CALL p(1, 'x');	Execute procedure

Reference habit: When learning a command, run it in a small database first, inspect the JSON response, then repeat through the driver or TLang program that will use it in production.

Part III

TLang Frontend Programming Language

26. Introducing TLang

TLang is the TigerDB frontend programming language. It is designed for practical database applications with screens, forms, menus, TigerDB connections, JSON results, RecordSets, and FOREACH iteration.

TLang Compilation Flow

A frontend language that produces standalone C applications

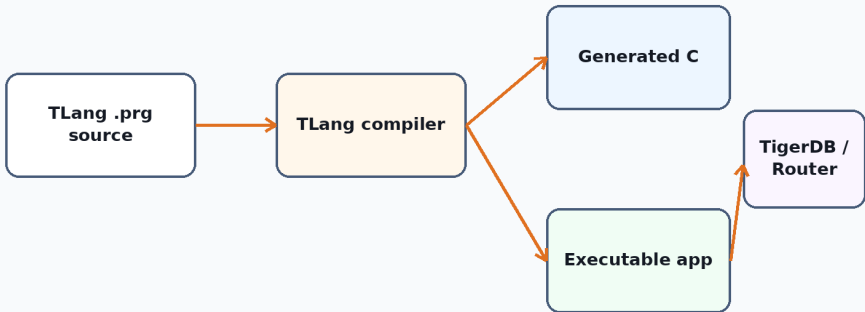


Figure. TLang compiles .prg programs into standalone C applications.

TLang is not only a scripting language for SQL. It is intended to help build frontend programs that can connect to TigerDB directly or through Router, display data, read user input, and call stored procedures.

The current stable line is Phase 4.4. It builds on Phase 4.3 RecordSet iteration and adds TigerDB frontend convenience methods such as PD.queryRS, PD.call, PD.callRS, PASSWORD ENV, OK, ERROR, and ENV.

TLang programs are compiled into C, and the generated executable includes the runtime support needed for values, strings, RecordSets, screen output, forms, TigerDB networking, and JSON conversion.

The language keeps syntax readable for business-style applications. It uses IF/ELSEIF/ELSE/ENDIF, WHILE/ENDWHILE, FOREACH/ENDFOREACH, and simple @ row,column SAY output.

Minimal TLang screen

```

SET SCREEN 24,80
CLEAR SCREEN
@1,1 SAY "TigerDB Frontend"
@3,1 SAY "Welcome to TLang"
  
```

27. TLang Program Structure

A TLang program is a sequence of declarations and commands. Variables hold values, functions encapsulate code, forms and menus organize interaction, and connection aliases such as PD represent TigerDB or Router sessions.

TLang uses var for flexible values and RecordSet for database result collections. The runtime value system supports strings, integers, doubles, booleans, nulls, connection values, RecordSets, and row references.

The compiler recognizes keywords such as SET, SCREEN, SAY, GET, CLEAR, IF, ELSE, ELSEIF, ENDIF, WHILE, ENDWHILE, FOREACH, ENDFOREACH, FUNCTION, RETURN, DEFINE, FORM, MENU, POPUP, CONNECT, TIGERDB, ROUTER, DATABASE, HOST, PORT, USER, and PASSWORD.

Code should be written in small sections: setup, connection, screen layout, query or procedure call, display, input, and close. This pattern makes generated C applications easier to test and reason about.

Because TLang compiles to C, program errors should be caught early. Start with a small .prg file, compile it, then add UI and database logic incrementally.

Variables and function structure

```
var title = "Customer List"
var rownum = 4

FUNCTION show_title()
    @1,1 SAY title
ENDFUNCTION

CALL show_title()
```

Construct	Syntax	Purpose
Variable	var x = 10	Store a runtime value
Function	FUNCTION name() ... ENDFUNCTION	Reusable TLang logic
RecordSet	RecordSet rs = ...	Rows returned by TigerDB
Connection alias	AS PD	Named TigerDB session

28. Variables, Expressions, and Control Flow

TLang control flow mirrors the practical structure used in TigerDB procedures but runs in the frontend application. IF and WHILE make applications interactive and dynamic.

Variables can store values from constants, expressions, input fields, function calls, database queries, and RecordSet rows. Clear variable names make frontend code easier to read than embedding values directly into SQL strings.

IF/ELSEIF/ELSE/ENDIF lets a program react to database results, user input, or environment settings. WHILE/ENDWHILE is useful for controlled loops. FOREACH is the preferred loop for rows because it directly expresses row-by-row processing.

Logical expressions should be kept simple. Complex business rules often belong in TigerDB stored procedures or in small named TLang functions rather than in one long condition.

TLang programs should close database connections before exiting. This is especially important when running through Router, where backend sessions and pooled connections may remain active until the Router session closes.

TLang IF/ELSEIF/ELSE/ENDIF

```
var amount = 750.25
var bucket = "small"

IF amount >= 1000 THEN
  bucket = "large"
ELSEIF amount >= 500 THEN
  bucket = "medium"
ELSE
  bucket = "small"
ENDIF

@5,1 SAY bucket
```

TLang WHILE loop

```
var i = 1
WHILE i <= 5 DO
  @i,1 SAY "Row " + i
  i = i + 1
ENDWHILE
```

29. Screens, Forms, Menus, and Tables

TLang includes frontend features for terminal-style applications: screen sizing, SAY output, GET input, forms, menus, popups, table borders, and masked input.

A TLang screen is a practical canvas for business applications. The @ row,column SAY pattern makes it simple to place labels and values. GET fields collect input and READ waits for the user interaction sequence.

Forms group inputs into a named area. Menus and popups let users select actions without remembering commands. Table borders help make reports readable in terminal-style output.

Masked input is useful for dates, amounts, and fixed-format values. It reduces mistakes before the program sends data to TigerDB.

When building a real application, keep the screen simple. A clear title, a few input fields, a response area, and a consistent command menu are more valuable than a crowded interface.

Simple input form layout

```
SET SCREEN 24,80
CLEAR SCREEN
DEFINE TABLE FROM 2,1 TO 8,78 DOUBLE
@3,3 SAY "Customer ID:"
@3,18 GET customer_id
@5,3 SAY "Amount:"
@5,18 GET amount
READ
```

UI design: Avoid putting too much text on one screen. Use menus for navigation and RecordSet screens for data display.

30. Connecting TLang to TigerDB

TLang can connect directly to TigerDB Server or to TigerDB Router. The connection statement supports host, port, user, password, optional database, and an alias used later for queries and method calls.

For direct server mode, the connection uses the TigerDB server port, normally 9191. For Router mode, the connection uses the Router port, normally 9292. The rest of the program can use the same alias style.

PASSWORD ENV lets a program avoid embedding a real password in source code or generated C. The runtime reads the password from an environment variable at execution time.

After connecting, the program can call PD.query for raw JSON, PD.queryRS for RecordSet conversion, PD.execute for SQL statements that do not need rows, and PD.close to close the session.

Connection errors should be displayed clearly. Use OK(json) and ERROR(json) helpers with raw JSON results when you need explicit success/failure handling.

Direct TigerDB connection

```
CONNECT TO TIGERDB SERVER
HOST "127.0.0.1" ON PORT 9191
USER "admin" PASSWORD ENV "TIGERDB_PASSWORD"
DATABASE "sales" AS PD
```

Router connection

```
CONNECT TO TIGERDB ROUTER
HOST "127.0.0.1" ON PORT 9292
USER "admin" PASSWORD ENV "TIGERDB_PASSWORD"
DATABASE "sales" AS PD
```

31. RecordSets and FOREACH

RecordSets make TigerDB JSON rows easy to use in TLang. A RecordSet can be created from `PD.queryRS`, `PD.callRS`, or `TO_RS(raw_json)`, then iterated with `FOREACH`.

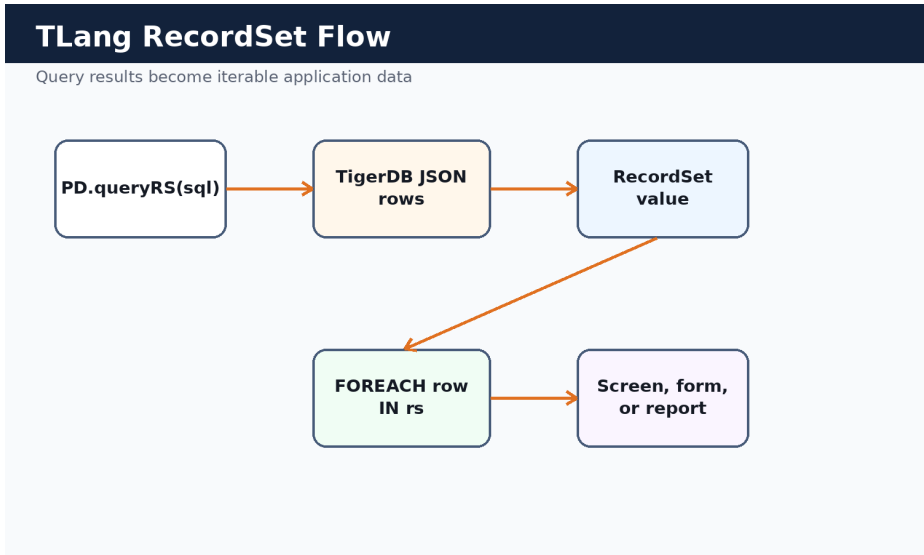


Figure. RecordSet values turn JSON rows into iterable frontend data.

`FOREACH row IN rs` expresses the most common database frontend task: for every returned row, display or process values. Row fields can be accessed with `row["field_name"]`.

`PD.queryRS` removes a common pattern. Instead of calling `PD.query` and then `TO_RS`, the program can ask for a RecordSet directly. This is clearer for readers and shorter in applications.

RecordSets are also useful for reports because they separate the database call from the display loop. A program can query once, then render the rows into a screen table, a form, or a print-style report.

When a query returns many rows, consider filtering or limiting the SQL. A frontend screen can only show a small number of rows at once, so the database should not send unnecessary data.

RecordSet display loop

```

RecordSet rs = PD.queryRS("SELECT id, name, amount FROM customers")
var rownum = 4

FOREACH row IN rs
  @rownum,3 SAY row["id"]
  @rownum,12 SAY row["name"]
  @rownum,40 SAY row["amount"]
  rownum = rownum + 1
ENDFOREACH
  
```


32. Calling Stored Procedures from TLang

TLang Phase 4.4 includes convenience methods for TigerDB stored procedures: PD.call returns raw JSON, and PD.callRS returns a RecordSet when the procedure returns rows.

Procedure calls are ideal for named business operations. The frontend program supplies parameters, while TigerDB validates the procedure catalog, checks the .sp file hash, applies invoker rights, and executes the procedure body.

PD.call is useful when a procedure performs work and the application wants to inspect the raw JSON success or error state. PD.callRS is useful when the procedure returns rows that should be displayed immediately.

TLang converts procedure arguments into SQL literals for CALL syntax. Strings are quoted and single quotes are escaped. Numeric and boolean values are passed in the expected format.

Because TigerDB blocks nested CALL inside procedures, TLang remains the correct place to orchestrate multiple procedure calls when a frontend workflow needs them.

TLang procedure helpers

```
var result = PD.call("record_sale", 1, 101, 750.25)

IF OK(result) THEN
    @10,3 SAY "Sale recorded"
ELSE
    @10,3 SAY "Error: " + ERROR(result)
ENDIF

RecordSet rows = PD.callRS("fetch_recent_sales")
```

33. A Complete TLang Application

A practical TLang application connects to TigerDB, retrieves rows, shows a screen, accepts input, calls a stored procedure, and closes the connection. The example below demonstrates the structure rather than every possible UI feature.

Begin by defining screen shape and collecting credentials through environment variables. This prevents passwords from becoming part of source files. Then connect to TigerDB or Router and select the database.

Next, use PD.queryRS to show existing data. If the user records a new sale, call a TigerDB stored procedure with parameters. The procedure keeps the business rule in the database while the TLang app focuses on interaction.

Finally, display the resulting rows and close the connection. If the application grows, split screen display, data loading, and procedure calls into named functions.

Complete TLang outline

```
SET SCREEN 24,80
CLEAR SCREEN
@1,3 SAY "TigerDB Sales Console"

CONNECT TO TIGERDB ROUTER HOST "127.0.0.1" ON PORT 9292
  USER "admin" PASSWORD ENV "TIGERDB_PASSWORD"
  DATABASE "sales" AS PD

RecordSet customers = PD.queryRS("SELECT id, name, amount FROM customers LIMIT 10")
var rownum = 4
FOREACH row IN customers
  @rownum,3 SAY row["id"]
  @rownum,12 SAY row["name"]
  @rownum,44 SAY row["amount"]
  rownum = rownum + 1
ENDFOREACH

var result = PD.call("record_sale", 1, 101, 750.25)
IF OK(result) THEN
  @18,3 SAY "Sale recorded successfully"
ELSE
  @18,3 SAY ERROR(result)
ENDIF

PD.close()
```

Part IV

TigerDB Router

34. Router Architecture

TigerDB Router is a gateway. It accepts the same length-prefixed JSON protocol used by TigerDB clients, authenticates Router users, maps database names to backend TigerDB servers, manages backend sessions and pools, and forwards requests.

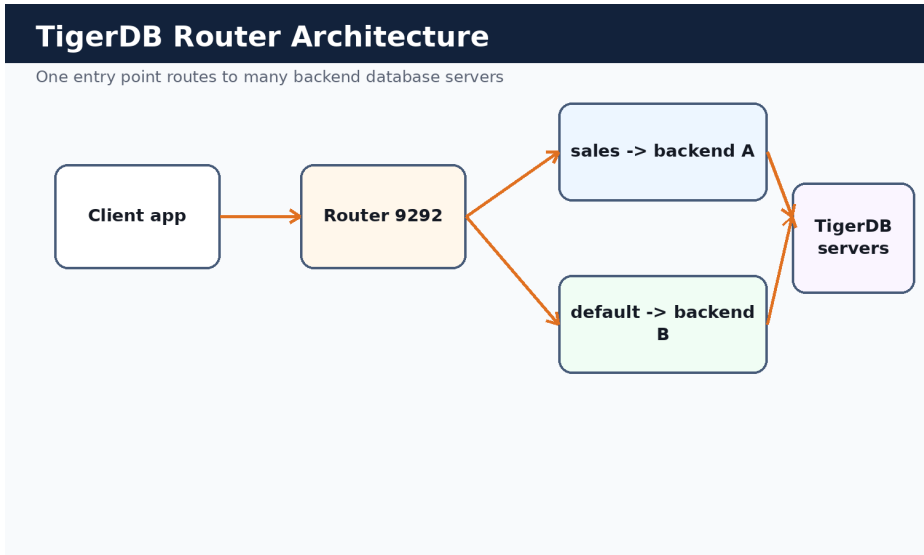


Figure. TigerDB Router gives applications one entry point while backend TigerDB servers remain private.

The Router is intentionally not a distributed query engine. It does not perform cross-server joins, it does not inspect stored-procedure files, and it does not execute procedure logic. The backend TigerDB Server owns SQL execution.

Router mode helps when applications should not know where every database lives. A database route in the config maps a database name to a backend server. Applications connect to Router and use the database name.

Backend pooling reduces repeated login overhead by reusing idle backend sessions when safe. Password-change synchronization, session visibility, health checks, and reload support make the Router practical for operations.

Because Router sees credentials and backend routes, it should run as a restricted OS user with protected config files. The backend TigerDB ports should normally be reachable only from the Router host or private network.

35. Router Configuration

Router configuration defines the listening host and port, max clients, backend pool settings, backend servers, database routes, users, grants, stored-procedure CALL policy, and authentication-throttling options.

A small Router configuration can route all databases to one backend. A larger configuration can route different databases to different backend hosts. Either way, applications use the Router endpoint rather than direct backend addresses.

The USER line contains the Router username and password and may also map the user to a backend TigerDB username and password. We currently rely on a dedicated OS user, file permissions, and firewall controls to protect this file.

GRANT lines define what a Router user can do for a database. READ permits read-oriented use, WRITE permits write-oriented use, and FULL permits administrative or definition operations depending on command classification.

Stored-procedure CALL policy is conservative by default. CALL can require FULL, or it can allow READ when same-backend-user mapping is enforced. This protects invoker-rights semantics when Router users map to backend users.

Router config example

```
LISTEN_HOST=0.0.0.0
LISTEN_PORT=9292
MAX_CLIENTS=256

BACKEND_POOL=ON
BACKEND_POOL_MAX_PER_BACKEND=16
BACKEND_POOL_IDLE_SECONDS=300

STORED_PROCEDURE_CALL_ROLE=FULL
STORED_PROCEDURE_CALL_REQUIRE_SAME_BACKEND_USER=ON

AUTH_THROTTLE=ON
AUTH_FAIL_WINDOW_SECONDS=60
AUTH_FAIL_THRESHOLD=5
AUTH_THROTTLE_SECONDS=5
AUTH_MAX_FAILURES_PER_SESSION=10

BACKEND main 127.0.0.1 9191
DATABASE default main
DATABASE sales main

USER admin admin1234 admin admin1234
GRANT admin * FULL
```

36. Routing Databases to Backends

Database routing is the core Router feature. Each DATABASE line maps a database name to a backend name, and each BACKEND line defines the backend host and port.

Routing lets the application ask for a database by name instead of by machine location. If the sales database later moves to another backend server, the Router configuration can be updated and reloaded without changing application code.

Router R3 introduced ROUTER RELOAD so many config changes can be applied without restart. LISTEN_HOST and LISTEN_PORT still require restart because the listening socket is already bound.

When a route changes, stale backend sessions reconnect lazily. This protects active sessions from raw pointer problems and ensures the Router can adopt updated backend information safely.

Health checks help administrators see whether a route points to a healthy backend, an unhealthy backend, or a missing backend definition.

Router Reload Without Restart

Bad config is rejected before replacing the running configuration

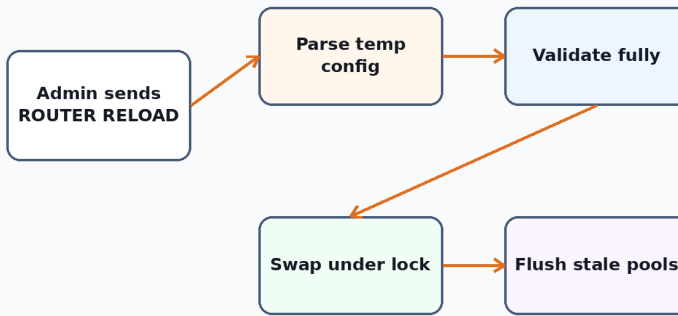


Figure. Router reload parses and validates a temporary config before replacing the running config.

Route multiple databases

```

BACKEND main 127.0.0.1 9191
BACKEND analytics 10.0.0.15 9191

DATABASE sales main
DATABASE inventory main
DATABASE reporting analytics

ROUTER RELOAD;
ROUTER DATABASES;
  
```


37. Router Users, Grants, and Stored Procedure Policy

Router users are application-facing identities. Grants decide what a Router user can do with a database route. Stored-procedure policy adds an extra guard around CALL forwarding.

Router permission checks happen before forwarding. For SQL, the Router classifies the command as read, write, or full. It then checks the user grant for the current database.

CREATE PROCEDURE and DROP PROCEDURE require FULL. SHOW PROCEDURES requires READ. CALL is configurable, with a conservative default of FULL. If CALL is relaxed to READ, same-backend-user enforcement should remain on.

The backend TigerDB server still performs its own authentication and authorization. Router checks should be treated as a front-line policy layer, not as a replacement for backend roles.

Normal users should not be mapped to backend admin credentials unless there is a deliberate and documented reason. Invoker-rights procedure behavior depends on the backend seeing the correct effective user.

Command	Router role	Reason
CREATE PROCEDURE	FULL	Defines reusable database logic
DROP PROCEDURE	FULL	Removes reusable database logic
SHOW PROCEDURES	READ	Metadata/reporting access
CALL	FULL by default	Avoid privilege confusion through backend mapping

Users and grants

```
USER report_user reportpass report_user reportpass
GRANT report_user sales READ

USER ingest_user ingestpass ingest_user ingestpass
GRANT ingest_user sales WRITE

USER app_admin apppass app_admin apppass
GRANT app_admin sales FULL
```

38. Router Health, Clients, and Reload

Router R5, R6, and R8 improved operational visibility and concurrency safety. Admin users can inspect backend health, session metadata, pools, database routes, and reload the config without restarting.

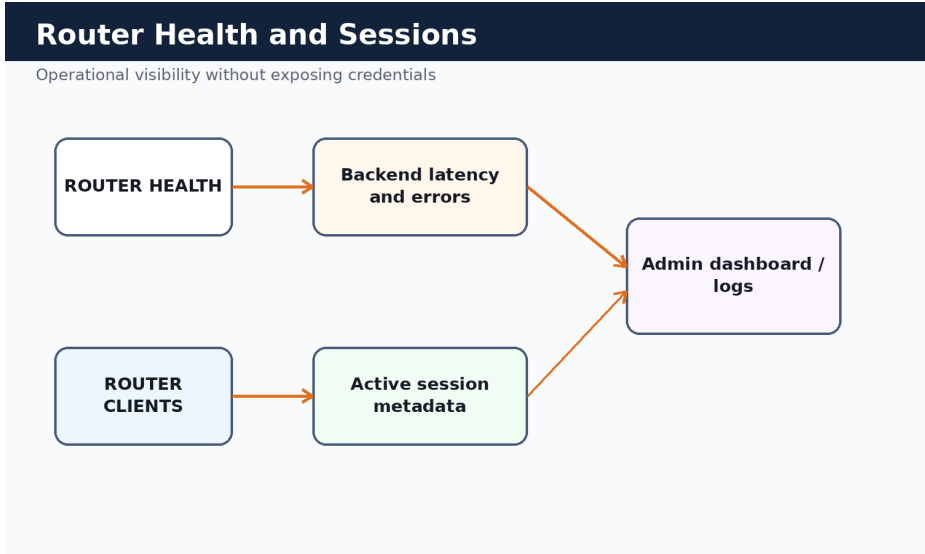


Figure. Router health and session commands provide operational visibility without exposing secrets.

ROUTER BACKENDS reports backend health, latency, check counts, consecutive failures, runtime failures, last errors, and idle pool connection counts. ROUTER HEALTH forces a live check, while ROUTER HEALTH STATUS returns cached health information.

ROUTER CLIENTS and ROUTER SESSIONS show safe active-session metadata such as peer, username, current database, backend name, idle seconds, request count, and last action. They do not include passwords.

ROUTER RELOAD parses a new config into a temporary object and validates it before replacing the running configuration. If parsing fails, the current config remains active.

R8 locking cleanup protects session snapshots, backend route snapshots, pool configuration reads, and health/status output while reload and health updates may be happening concurrently.

Router admin commands

```

ROUTER STATUS;
ROUTER BACKENDS;
ROUTER DATABASES;
ROUTER POOLS;
ROUTER HEALTH;
ROUTER HEALTH STATUS;
ROUTER CLIENTS;
ROUTER RELOAD;
  
```


39. Router Security and Operations

Router security combines credentials, grants, auth throttling, socket timeouts, backend session management, safe reload, and operating-system controls. The Router should be treated as a controlled gateway.

Authentication throttling tracks failed login attempts by client IP within a window. When a client exceeds the threshold, Router returns a retry-after response instead of sleeping and tying up worker threads.

Socket timeouts and keepalive settings reduce the risk of idle or slow clients consuming resources indefinitely. Backend sessions are closed or revalidated when identity, generation, or database state changes.

The current deployment decision is to rely on a dedicated tigerdb OS user, 600 config permissions, 700 config directory permissions, a restrictive service umask, and firewall isolation rather than adding password encryption into Router config.

Operationally, Router should be monitored with ROUTER STATUS, ROUTER BACKENDS, ROUTER HEALTH STATUS, ROUTER CLIENTS, and system logs. Config changes should be tested with ROUTER RELOAD before a maintenance window.

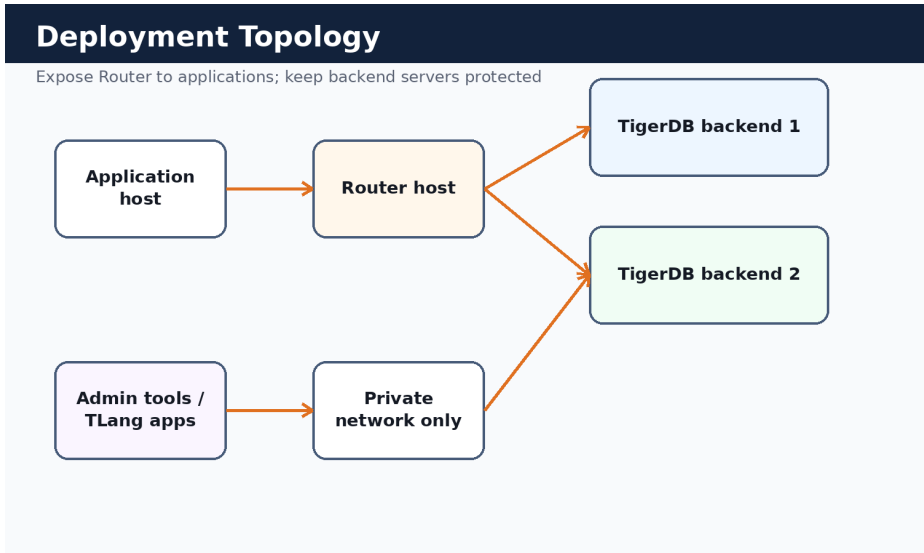


Figure. Recommended deployment exposes Router to applications and protects backend database servers.

Operating-system protection example

```

sudo useradd --system --home /var/lib/tigerdb --shell /usr/sbin/nologin tigerdb
sudo chown tigerdb:tigerdb /etc/tigerdb-router/tigerdb_router.config
sudo chmod 600 /etc/tigerdb-router/tigerdb_router.config
sudo chmod 700 /etc/tigerdb-router

```

Part V

Appendices and Operational Examples

40. TigerDB Server Configuration Examples

The TigerDB server configuration controls worker counts, memory, auto vacuum, WAL behavior, auto tuning, licensing, and paths. The examples below should be adapted to the machine and workload.

Balanced TigerDB configuration

```
# tigerdb.config - balanced development example
READ_WORKERS=4
WRITE_WORKERS=2
CONNECTION_WORKERS=8
MEMORY_LIMIT_MB=1024
MEMORY_FLUSH_PERCENT=80
MEMORY_HARD_STOP_PERCENT=95
AUTO_TUNE=ON

AUTO_VACUUM=ON
AUTO_VACUUM_IDLE_SECONDS=10
AUTO_VACUUM_INTERVAL_SECONDS=60
AUTO_VACUUM_MEMORY_PERCENT=70
AUTO_VACUUM_WAL_MB=64

WAL_SYNC_EVERY=16
COMMIT_GROUP_WAIT_MS=2

LICENSE_KEY=your-license-key
LICENSE_SERVER_HOST=159.223.89.223
LICENSE_SERVER_PORT=9443
LICENSE_SERVER_TIMEOUT_SECONDS=10
LICENSE_CHECK_INTERVAL_SECONDS=3600
LICENSE_CACHE_FILE=/tigerdb_data/license/tigerdb.license
```

For benchmark-only runs, it can be useful to temporarily disable auto vacuum to remove background maintenance from timing. Production deployments should normally keep maintenance enabled unless there is a tested operational alternative.

AUTO_TUNE lets the server pick sensible defaults based on CPU, memory, and WAL fsync behavior. Explicit settings override auto-tuned values when the corresponding config flag is set.

WAL settings should be chosen with durability and throughput in mind. A lower WAL_SYNC_EVERY value can increase sync frequency; a higher value can improve throughput but should be evaluated against durability requirements.

License fields must be set before normal database actions are allowed. PING and SHOW LICENSE are useful for diagnosing license state before application testing.

Setting	Meaning	Common note
READ_WORKERS	Read worker count	Often CPU dependent
WRITE_WORKERS	Write worker count	Avoid excessive contention

MEMORY_LIMIT_MB	Memory target	Set lower on small hosts
AUTO_VACUUM	Enable maintenance	Useful for WAL/table cleanup
WAL_SYNC_EVERY	Sync grouping	Benchmark with care
LICENSE_KEY	Activation key	Required for normal use

41. Router Configuration Examples

Router configuration is line-oriented and practical. It defines listener behavior, backend pool settings, stored-procedure policy, auth throttling, backends, database routes, users, and grants.

Single-backend Router config

```
# tigerdb_router.config - production-style single backend
LISTEN_HOST=0.0.0.0
LISTEN_PORT=9292
MAX_CLIENTS=256

BACKEND_POOL=ON
BACKEND_POOL_MAX_PER_BACKEND=32
BACKEND_POOL_IDLE_SECONDS=300

STORED_PROCEDURE_CALL_ROLE=FULL
STORED_PROCEDURE_CALL_REQUIRE_SAME_BACKEND_USER=ON

AUTH_THROTTLE=ON
AUTH_FAIL_WINDOW_SECONDS=60
AUTH_FAIL_THRESHOLD=5
AUTH_THROTTLE_SECONDS=5
AUTH_MAX_FAILURES_PER_SESSION=10

BACKEND main 127.0.0.1 9191
DATABASE default main
DATABASE sales main
DATABASE inventory main

ADMIN admin admin1234 admin admin1234
GRANT admin * FULL

USER report reportpass report reportpass
GRANT report sales READ

USER ingest ingestpass ingest ingestpass
GRANT ingest sales WRITE
```

Multi-backend routing

```
# multi-backend route example
BACKEND main 10.0.0.10 9191
BACKEND analytics 10.0.0.20 9191
BACKEND archive 10.0.0.30 9191

DATABASE sales main
DATABASE inventory main
DATABASE reporting analytics
DATABASE old_sales archive
```

Router config files contain sensitive credentials in the current approach. Protect them with a dedicated OS user, 600 file permissions, 700 directory permissions, service umask 0077, backups with the same restrictions, and firewall rules.

When changing backends or routes, use ROUTER RELOAD from an admin session. If LISTEN_HOST or LISTEN_PORT changes, restart is still required.

If CALL is relaxed from FULL to READ, keep same-backend-user enforcement enabled and avoid mapping normal users to backend administrator accounts.

ROUTER HEALTH STATUS and ROUTER DATABASES should be checked after route changes to confirm that every database points to a healthy backend.

42. Command Cookbook

The cookbook collects complete command sequences that can be copied into a client, adapted, and tested. Each sequence is intentionally small and self-contained.

Cookbook: first database

```
-- Create and populate a small sales database
LOGIN admin IDENTIFIED BY 'admin1234';
CREATE DATABASE sales;
USE sales;
CREATE TABLE customers (id INT PRIMARY KEY, name STRING, city STRING, amount DOUBLE);
INSERT INTO customers VALUES (1, 'Anil', 'Pune', 250.75);
INSERT INTO customers VALUES (2, 'Neha', 'Mumbai', 500.00);
SELECT * FROM customers;
```

Cookbook: index lookup

```
-- Add an index and run a lookup
CREATE INDEX idx_customers_city ON customers(city);
SHOW INDEXES ON customers;
SELECT id, name, city FROM customers WHERE city = 'Mumbai';
```

Cookbook: parameterized procedure

```
-- Create and call a stored procedure
CREATE PROCEDURE get_customer
ON customers
(IN p_id INT)
AS $$
SELECT id, name, city, amount FROM customers WHERE id = :p_id
$$;

CALL get_customer(1);
```

Cookbook: Router checks

```
-- Router admin checks
ROUTER STATUS;
ROUTER BACKENDS;
ROUTER DATABASES;
ROUTER HEALTH STATUS;
ROUTER CLIENTS;
```

Cookbook: TLang display

```
-- TLang query and display
CONNECT TO TIGERDB ROUTER HOST "127.0.0.1" ON PORT 9292
USER "admin" PASSWORD ENV "TIGERDB_PASSWORD"
DATABASE "sales" AS PD

RecordSet rs = PD.queryRS("SELECT id, name, amount FROM customers")
FOREACH row IN rs
    @3,3 SAY row["name"]
ENDFOREACH
PD.close()
```

43. Troubleshooting Checklist

Troubleshooting is easier when questions are asked in order: Is the process running? Is the port reachable? Is the license valid? Is the user authenticated? Is the current database correct? Is the command authorized?

Symptom	Likely area	First check
PING fails	Network/server process	Is TigerDB running on port 9191?
SHOW LICENSE says MISSING	License config	Check LICENSE_KEY in tigerdb.config
Login times out	Server readiness/network	Try PING and restart if process is stuck
Password must be changed	User state	Run CHANGE PASSWORD TO ...
Permission denied	Role/grant	Check user role or Router GRANT
Large INSERT fails through Router	Router large SQL support	Use R2+ Router line
CALL denied	Procedure policy/permissions	Check Router CALL role and backend user mapping
Procedure hash mismatch	Tampered .sp file	Recreate procedure through approved command
ORDER BY slow	Sort path/result size	Benchmark fullscan vs order-by
Indexed insert slow	Index maintenance	Measure indexed and unindexed separately

For direct TigerDB issues, start with PING, SHOW LICENSE, LOGIN, SHOW DATABASES, and a one-row SELECT. For Router issues, start with ROUTER STATUS, ROUTER BACKENDS, ROUTER DATABASES, ROUTER HEALTH STATUS, and ROUTER CLIENTS.

If a benchmark fails during login, it is not an INSERT parser problem. It means the server did not respond to the initial authentication request in time. Check whether another benchmark or background maintenance is occupying the server.

If large batch INSERT works directly but fails through Router, verify that the Router binary includes the R2 large SQL forwarding improvement. Older Router versions used an 8192-byte SQL extraction buffer.

If a procedure fails, check SHOW PROCEDURES. It can reveal whether the stored .sp file is enabled and whether the hash is valid. Tampered files should be denied until the procedure is recreated or restored through the proper path.

44. Glossary

The glossary summarizes terms used throughout the book. These definitions are practical rather than academic, and they are written in the context of TigerDB, TLang, and Router.

Term	Meaning
Backend	A TigerDB Server instance behind the Router.
Catalog	Metadata describing databases, tables, roles, or procedures.
Checkpoint	A durable state marker used with WAL for recovery.
Column	A named value slot inside a table.
Config database	TigerDB internal database used for users, roles, and procedure metadata.
Database route	Router mapping from database name to backend.
FOREACH	TLang loop for iterating RecordSet rows.
Invoker rights	Procedure execution using the caller's permissions.
JSON protocol	Length-prefixed request/response format used by TigerDB clients.
Primary key	Column or value that uniquely identifies a row.
RecordSet	TLang runtime value representing rows returned by TigerDB.
Router grant	Permission assigned to a Router user for a database.
Stored procedure	Named TigerDB .sp script with internal trusted metadata.
TLang	TigerDB frontend programming language.
WAL	Write-ahead log used to make changes durable.

Vocabulary matters because it shortens conversations. A team that understands the difference between a Router user and a backend TigerDB user can discuss permissions more accurately.

Likewise, a team that understands WAL, checkpoint, and stored-procedure hash validation will diagnose failures faster and avoid changing the wrong part of the system.

The safest operating practice is to keep diagrams, configuration files, and command examples together. A new developer should be able to read the configuration, run the smoke tests, and understand the data flow from application to Router to TigerDB Server.

45. Source and Version Notes

This book is written from the current TigerDB project context and the stable baselines established during development. It is intended as user and administrator documentation, not as a source-code internals manual.

TigerDB Server stable baseline:

`tigerdb_v5_phase_sp4c_performance_r4c_stable_order_by.c`. This line preserves stored procedures through WHILE/ENDWHILE, large SQL batch handling, stable write and read performance, and safe ORDER BY improvement.

TLang stable baseline: Tiger Language Phase 4.4 frontend convenience upgrade. This line adds PD.queryRS, PD.call, PD.callRS, PASSWORD ENV, ENV, OK, and ERROR while preserving RecordSet and FOREACH support.

TigerDB Router stable baseline: `tigerdb_router_v10_r8_config_health_locking.c`. This line includes large SQL forwarding, config reload, backend health/status, session visibility, auth throttling, and config/health locking cleanup.

The examples in this book use practical command forms that should be tested against the exact binary being deployed. Future versions may add features, but this edition treats the baselines above as protected stable lines.

Documentation rule: When a new experimental branch is created, update the book only after the feature is promoted to a stable baseline and tested with smoke, security, and performance checks.

Function quick reference

Function	Category	Example
COUNT	Aggregate	SELECT COUNT(*) FROM sales;
SUM	Aggregate	SELECT SUM(amount) FROM sales;
AVG/MEAN	Aggregate	SELECT AVG(amount) FROM sales;
MIN/MAX	Aggregate	SELECT MIN(amount), MAX(amount) FROM sales;
MED/MEDIAN	Aggregate	SELECT MEDIAN(amount) FROM sales;
VAR	Statistics	SELECT VAR(amount) FROM sales;
STDDEV	Statistics	SELECT STDDEV(amount) FROM sales;
STDDEV_POP	Statistics	SELECT STDDEV_POP(amount) FROM sales;
STDDEV_SAMP	Statistics	SELECT STDDEV_SAMP(amount) FROM sales;
ABS	Scalar	SELECT ABS(amount) FROM sales;
ROUND	Scalar	SELECT ROUND(amount) FROM sales;
FLOOR	Scalar	SELECT FLOOR(amount) FROM sales;
CEIL/CEILING	Scalar	SELECT CEIL(amount) FROM sales;
SQRT	Scalar	SELECT SQRT(amount) FROM sales;
LOG/LN/LOG10	Scalar	SELECT LOG(amount) FROM sales;
EXP	Scalar	SELECT EXP(amount) FROM sales;
POW	Scalar	SELECT POW(amount, 2) FROM sales;
SMA/EMA/WMA	Ordered analytics	SELECT SMA(close,20) FROM ticks ORDER BY ts;
ROC/MOMENTUM	Ordered analytics	SELECT ROC(close,10) FROM ticks ORDER BY ts;
LOGRET/VOLATILITY/RSI	Market analytics	SELECT RSI(close,14) FROM ticks ORDER BY ts;

Function results are only as meaningful as the query that feeds them. For aggregate functions, confirm the set of rows being summarized. For ordered analytics, confirm that ORDER BY expresses the correct time sequence.

When building reports, prefer clear aliases in application code even if the database result label is already readable. Good naming reduces confusion when a report combines raw values and calculated columns.

For financial-style functions, test with a small sample table before using live market data. The earliest rows may not have enough history to compute a value for a given period.

Part VI

Extended Workbook, Examples, and Operational Playbooks

Workbook 1. Create a database and select it

This workbook lesson turns the idea of database creation and context into a repeatable exercise. The goal is not to memorize a command, but to understand what changes in the database, what the response should look like, and what the operator should check before moving to a larger data set.

Begin by reading the example slowly. Identify the database, the table, the key columns, the command that changes state, and the command that verifies the result. This habit is useful for TigerDB, TLang applications, and Router deployments because all three systems reward clear state transitions.

When practicing database creation and context, use a clean database or a table name that will not conflict with production data. Run every command once, inspect the JSON response, and repeat the same operation through the driver or frontend program that will eventually use it.

Exercise: Create a database and select it

```
LOGIN admin IDENTIFIED BY 'admin1234';
CREATE DATABASE workbook_sales;
USE workbook_sales;
SHOW DATABASES;
```

Expected learning outcome

- You can explain what database creation and context means in ordinary language.
- You can identify the exact command that creates, changes, or reads state.
- You can inspect the response and decide whether the operation succeeded.
- You can name one security or performance consideration before production use.

Practice notes

Repeat the example with one changed value. Then deliberately make a small mistake, such as using a missing table or an unauthorized user, and observe the error. Understanding failure responses is as important as understanding successful commands.

Record the command sequence in a team notebook or deployment guide. A short, tested sequence is more valuable than a long memory of how the system probably behaves.

Workbook 2. Create a table with a primary key

This workbook lesson turns the idea of table identity into a repeatable exercise. The goal is not to memorize a command, but to understand what changes in the database, what the response should look like, and what the operator should check before moving to a larger data set.

Begin by reading the example slowly. Identify the database, the table, the key columns, the command that changes state, and the command that verifies the result. This habit is useful for TigerDB, TLang applications, and Router deployments because all three systems reward clear state transitions.

When practicing table identity, use a clean database or a table name that will not conflict with production data. Run every command once, inspect the JSON response, and repeat the same operation through the driver or frontend program that will eventually use it.

Exercise: Create a table with a primary key

```
CREATE TABLE customers (  
  id INT PRIMARY KEY,  
  name STRING,  
  amount DOUBLE,  
  active BOOL  
);  
SHOW TABLE customers;
```

Expected learning outcome

- You can explain what table identity means in ordinary language.
- You can identify the exact command that creates, changes, or reads state.
- You can inspect the response and decide whether the operation succeeded.
- You can name one security or performance consideration before production use.

Practice notes

Repeat the example with one changed value. Then deliberately make a small mistake, such as using a missing table or an unauthorized user, and observe the error. Understanding failure responses is as important as understanding successful commands.

Record the command sequence in a team notebook or deployment guide. A short, tested sequence is more valuable than a long memory of how the system probably behaves.

Workbook 3. Insert and verify rows

This workbook lesson turns the idea of durable row creation into a repeatable exercise. The goal is not to memorize a command, but to understand what changes in the database, what the response should look like, and what the operator should check before moving to a larger data set.

Begin by reading the example slowly. Identify the database, the table, the key columns, the command that changes state, and the command that verifies the result. This habit is useful for TigerDB, TLang applications, and Router deployments because all three systems reward clear state transitions.

When practicing durable row creation, use a clean database or a table name that will not conflict with production data. Run every command once, inspect the JSON response, and repeat the same operation through the driver or frontend program that will eventually use it.

Exercise: Insert and verify rows

```
INSERT INTO customers VALUES (1, 'Anil', 250.75, true);  
INSERT INTO customers VALUES (2, 'Neha', 500.00, true);  
SELECT * FROM customers;
```

Expected learning outcome

- You can explain what durable row creation means in ordinary language.
- You can identify the exact command that creates, changes, or reads state.
- You can inspect the response and decide whether the operation succeeded.
- You can name one security or performance consideration before production use.

Practice notes

Repeat the example with one changed value. Then deliberately make a small mistake, such as using a missing table or an unauthorized user, and observe the error. Understanding failure responses is as important as understanding successful commands.

Record the command sequence in a team notebook or deployment guide. A short, tested sequence is more valuable than a long memory of how the system probably behaves.

Workbook 4. Update carefully

This workbook lesson turns the idea of safe modification into a repeatable exercise. The goal is not to memorize a command, but to understand what changes in the database, what the response should look like, and what the operator should check before moving to a larger data set.

Begin by reading the example slowly. Identify the database, the table, the key columns, the command that changes state, and the command that verifies the result. This habit is useful for TigerDB, TLang applications, and Router deployments because all three systems reward clear state transitions.

When practicing safe modification, use a clean database or a table name that will not conflict with production data. Run every command once, inspect the JSON response, and repeat the same operation through the driver or frontend program that will eventually use it.

Exercise: Update carefully

```
SELECT * FROM customers WHERE id = 2;
UPDATE customers SET amount = 625.00 WHERE id = 2;
SELECT * FROM customers WHERE id = 2;
```

Expected learning outcome

- You can explain what safe modification means in ordinary language.
- You can identify the exact command that creates, changes, or reads state.
- You can inspect the response and decide whether the operation succeeded.
- You can name one security or performance consideration before production use.

Practice notes

Repeat the example with one changed value. Then deliberately make a small mistake, such as using a missing table or an unauthorized user, and observe the error. Understanding failure responses is as important as understanding successful commands.

Record the command sequence in a team notebook or deployment guide. A short, tested sequence is more valuable than a long memory of how the system probably behaves.

Workbook 5. Delete carefully

This workbook lesson turns the idea of controlled deletion into a repeatable exercise. The goal is not to memorize a command, but to understand what changes in the database, what the response should look like, and what the operator should check before moving to a larger data set.

Begin by reading the example slowly. Identify the database, the table, the key columns, the command that changes state, and the command that verifies the result. This habit is useful for TigerDB, TLang applications, and Router deployments because all three systems reward clear state transitions.

When practicing controlled deletion, use a clean database or a table name that will not conflict with production data. Run every command once, inspect the JSON response, and repeat the same operation through the driver or frontend program that will eventually use it.

Exercise: Delete carefully

```
SELECT * FROM customers WHERE id = 2;
DELETE FROM customers WHERE id = 2;
SELECT * FROM customers;
```

Expected learning outcome

- You can explain what controlled deletion means in ordinary language.
- You can identify the exact command that creates, changes, or reads state.
- You can inspect the response and decide whether the operation succeeded.
- You can name one security or performance consideration before production use.

Practice notes

Repeat the example with one changed value. Then deliberately make a small mistake, such as using a missing table or an unauthorized user, and observe the error. Understanding failure responses is as important as understanding successful commands.

Record the command sequence in a team notebook or deployment guide. A short, tested sequence is more valuable than a long memory of how the system probably behaves.

Workbook 6. Create and inspect an index

This workbook lesson turns the idea of indexed lookup into a repeatable exercise. The goal is not to memorize a command, but to understand what changes in the database, what the response should look like, and what the operator should check before moving to a larger data set.

Begin by reading the example slowly. Identify the database, the table, the key columns, the command that changes state, and the command that verifies the result. This habit is useful for TigerDB, TLang applications, and Router deployments because all three systems reward clear state transitions.

When practicing indexed lookup, use a clean database or a table name that will not conflict with production data. Run every command once, inspect the JSON response, and repeat the same operation through the driver or frontend program that will eventually use it.

Exercise: Create and inspect an index

```
CREATE INDEX idx_customers_active ON customers(active);  
SHOW INDEXES ON customers;  
SELECT * FROM customers WHERE active = true;
```

Expected learning outcome

- You can explain what indexed lookup means in ordinary language.
- You can identify the exact command that creates, changes, or reads state.
- You can inspect the response and decide whether the operation succeeded.
- You can name one security or performance consideration before production use.

Practice notes

Repeat the example with one changed value. Then deliberately make a small mistake, such as using a missing table or an unauthorized user, and observe the error. Understanding failure responses is as important as understanding successful commands.

Record the command sequence in a team notebook or deployment guide. A short, tested sequence is more valuable than a long memory of how the system probably behaves.

Workbook 7. Group rows into a report

This workbook lesson turns the idea of aggregation into a repeatable exercise. The goal is not to memorize a command, but to understand what changes in the database, what the response should look like, and what the operator should check before moving to a larger data set.

Begin by reading the example slowly. Identify the database, the table, the key columns, the command that changes state, and the command that verifies the result. This habit is useful for TigerDB, TLang applications, and Router deployments because all three systems reward clear state transitions.

When practicing aggregation, use a clean database or a table name that will not conflict with production data. Run every command once, inspect the JSON response, and repeat the same operation through the driver or frontend program that will eventually use it.

Exercise: Group rows into a report

```
INSERT INTO customers VALUES (3, 'Kiran', 125.50, false);
SELECT active, COUNT(*), SUM(amount), AVG(amount)
FROM customers
GROUP BY active;
```

Expected learning outcome

- You can explain what aggregation means in ordinary language.
- You can identify the exact command that creates, changes, or reads state.
- You can inspect the response and decide whether the operation succeeded.
- You can name one security or performance consideration before production use.

Practice notes

Repeat the example with one changed value. Then deliberately make a small mistake, such as using a missing table or an unauthorized user, and observe the error. Understanding failure responses is as important as understanding successful commands.

Record the command sequence in a team notebook or deployment guide. A short, tested sequence is more valuable than a long memory of how the system probably behaves.

Workbook 8. Sort and limit results

This workbook lesson turns the idea of ordered reporting into a repeatable exercise. The goal is not to memorize a command, but to understand what changes in the database, what the response should look like, and what the operator should check before moving to a larger data set.

Begin by reading the example slowly. Identify the database, the table, the key columns, the command that changes state, and the command that verifies the result. This habit is useful for TigerDB, TLang applications, and Router deployments because all three systems reward clear state transitions.

When practicing ordered reporting, use a clean database or a table name that will not conflict with production data. Run every command once, inspect the JSON response, and repeat the same operation through the driver or frontend program that will eventually use it.

Exercise: Sort and limit results

```
SELECT id, name, amount
FROM customers
ORDER BY amount DESC
LIMIT 5;
```

Expected learning outcome

- You can explain what ordered reporting means in ordinary language.
- You can identify the exact command that creates, changes, or reads state.
- You can inspect the response and decide whether the operation succeeded.
- You can name one security or performance consideration before production use.

Practice notes

Repeat the example with one changed value. Then deliberately make a small mistake, such as using a missing table or an unauthorized user, and observe the error. Understanding failure responses is as important as understanding successful commands.

Record the command sequence in a team notebook or deployment guide. A short, tested sequence is more valuable than a long memory of how the system probably behaves.

Workbook 9. Create a simple procedure

This workbook lesson turns the idea of stored procedure registration into a repeatable exercise. The goal is not to memorize a command, but to understand what changes in the database, what the response should look like, and what the operator should check before moving to a larger data set.

Begin by reading the example slowly. Identify the database, the table, the key columns, the command that changes state, and the command that verifies the result. This habit is useful for TigerDB, TLang applications, and Router deployments because all three systems reward clear state transitions.

When practicing stored procedure registration, use a clean database or a table name that will not conflict with production data. Run every command once, inspect the JSON response, and repeat the same operation through the driver or frontend program that will eventually use it.

Exercise: Create a simple procedure

```
CREATE PROCEDURE list_customers ON customers AS $$  
SELECT id, name, amount FROM customers  
$$;  
SHOW PROCEDURES ON customers;  
CALL list_customers();
```

Expected learning outcome

- You can explain what stored procedure registration means in ordinary language.
- You can identify the exact command that creates, changes, or reads state.
- You can inspect the response and decide whether the operation succeeded.
- You can name one security or performance consideration before production use.

Practice notes

Repeat the example with one changed value. Then deliberately make a small mistake, such as using a missing table or an unauthorized user, and observe the error. Understanding failure responses is as important as understanding successful commands.

Record the command sequence in a team notebook or deployment guide. A short, tested sequence is more valuable than a long memory of how the system probably behaves.

Workbook 10. Use procedure parameters

This workbook lesson turns the idea of procedure input validation into a repeatable exercise. The goal is not to memorize a command, but to understand what changes in the database, what the response should look like, and what the operator should check before moving to a larger data set.

Begin by reading the example slowly. Identify the database, the table, the key columns, the command that changes state, and the command that verifies the result. This habit is useful for TigerDB, TLang applications, and Router deployments because all three systems reward clear state transitions.

When practicing procedure input validation, use a clean database or a table name that will not conflict with production data. Run every command once, inspect the JSON response, and repeat the same operation through the driver or frontend program that will eventually use it.

Exercise: Use procedure parameters

```
CREATE PROCEDURE get_customer ON customers (IN p_id INT) AS $$
SELECT id, name, amount FROM customers WHERE id = :p_id
$$;
CALL get_customer(1);
```

Expected learning outcome

- You can explain what procedure input validation means in ordinary language.
- You can identify the exact command that creates, changes, or reads state.
- You can inspect the response and decide whether the operation succeeded.
- You can name one security or performance consideration before production use.

Practice notes

Repeat the example with one changed value. Then deliberately make a small mistake, such as using a missing table or an unauthorized user, and observe the error. Understanding failure responses is as important as understanding successful commands.

Record the command sequence in a team notebook or deployment guide. A short, tested sequence is more valuable than a long memory of how the system probably behaves.

Workbook 11. Use local variables

This workbook lesson turns the idea of procedure state into a repeatable exercise. The goal is not to memorize a command, but to understand what changes in the database, what the response should look like, and what the operator should check before moving to a larger data set.

Begin by reading the example slowly. Identify the database, the table, the key columns, the command that changes state, and the command that verifies the result. This habit is useful for TigerDB, TLang applications, and Router deployments because all three systems reward clear state transitions.

When practicing procedure state, use a clean database or a table name that will not conflict with production data. Run every command once, inspect the JSON response, and repeat the same operation through the driver or frontend program that will eventually use it.

Exercise: Use local variables

```
CREATE PROCEDURE insert_customer ON customers
(IN p_id INT, IN p_name STRING, IN p_amount DOUBLE) AS $$
DECLARE v_active BOOL DEFAULT true;
INSERT INTO customers VALUES (:p_id, :p_name, :p_amount, :v_active);
SELECT * FROM customers WHERE id = :p_id
$$;
```

Expected learning outcome

- You can explain what procedure state means in ordinary language.
- You can identify the exact command that creates, changes, or reads state.
- You can inspect the response and decide whether the operation succeeded.
- You can name one security or performance consideration before production use.

Practice notes

Repeat the example with one changed value. Then deliberately make a small mistake, such as using a missing table or an unauthorized user, and observe the error. Understanding failure responses is as important as understanding successful commands.

Record the command sequence in a team notebook or deployment guide. A short, tested sequence is more valuable than a long memory of how the system probably behaves.

Workbook 12. Use IF logic

This workbook lesson turns the idea of business rules into a repeatable exercise. The goal is not to memorize a command, but to understand what changes in the database, what the response should look like, and what the operator should check before moving to a larger data set.

Begin by reading the example slowly. Identify the database, the table, the key columns, the command that changes state, and the command that verifies the result. This habit is useful for TigerDB, TLang applications, and Router deployments because all three systems reward clear state transitions.

When practicing business rules, use a clean database or a table name that will not conflict with production data. Run every command once, inspect the JSON response, and repeat the same operation through the driver or frontend program that will eventually use it.

Exercise: Use IF logic

```
CREATE PROCEDURE classify_amount ON customers
(IN p_id INT, IN p_amount DOUBLE) AS $$
DECLARE v_active BOOL DEFAULT true;
IF :p_amount >= 500 THEN
SET v_active = true;
ELSE
SET v_active = false;
ENDIF;
UPDATE customers SET active = :v_active WHERE id = :p_id;
SELECT * FROM customers WHERE id = :p_id
$$;
```

Expected learning outcome

- You can explain what business rules means in ordinary language.
- You can identify the exact command that creates, changes, or reads state.
- You can inspect the response and decide whether the operation succeeded.
- You can name one security or performance consideration before production use.

Practice notes

Repeat the example with one changed value. Then deliberately make a small mistake, such as using a missing table or an unauthorized user, and observe the error. Understanding failure responses is as important as understanding successful commands.

Record the command sequence in a team notebook or deployment guide. A short, tested sequence is more valuable than a long memory of how the system probably behaves.

Workbook 13. Use WHILE logic

This workbook lesson turns the idea of controlled repetition into a repeatable exercise. The goal is not to memorize a command, but to understand what changes in the database, what the response should look like, and what the operator should check before moving to a larger data set.

Begin by reading the example slowly. Identify the database, the table, the key columns, the command that changes state, and the command that verifies the result. This habit is useful for TigerDB, TLang applications, and Router deployments because all three systems reward clear state transitions.

When practicing controlled repetition, use a clean database or a table name that will not conflict with production data. Run every command once, inspect the JSON response, and repeat the same operation through the driver or frontend program that will eventually use it.

Exercise: Use WHILE logic

```
CREATE TABLE numbers (id INT PRIMARY KEY, label STRING);
CREATE PROCEDURE seed_numbers ON numbers (IN p_limit INT) AS $$
DECLARE v_i INT DEFAULT 1;
WHILE :v_i <= :p_limit DO
INSERT INTO numbers VALUES (:v_i, 'generated');
SET v_i = :v_i + 1;
ENDWHILE;
SELECT * FROM numbers
$$;
CALL seed_numbers(5);
```

Expected learning outcome

- You can explain what controlled repetition means in ordinary language.
- You can identify the exact command that creates, changes, or reads state.
- You can inspect the response and decide whether the operation succeeded.
- You can name one security or performance consideration before production use.

Practice notes

Repeat the example with one changed value. Then deliberately make a small mistake, such as using a missing table or an unauthorized user, and observe the error. Understanding failure responses is as important as understanding successful commands.

Record the command sequence in a team notebook or deployment guide. A short, tested sequence is more valuable than a long memory of how the system probably behaves.

Workbook 14. Check license state

This workbook lesson turns the idea of license diagnostics into a repeatable exercise. The goal is not to memorize a command, but to understand what changes in the database, what the response should look like, and what the operator should check before moving to a larger data set.

Begin by reading the example slowly. Identify the database, the table, the key columns, the command that changes state, and the command that verifies the result. This habit is useful for TigerDB, TLang applications, and Router deployments because all three systems reward clear state transitions.

When practicing license diagnostics, use a clean database or a table name that will not conflict with production data. Run every command once, inspect the JSON response, and repeat the same operation through the driver or frontend program that will eventually use it.

Exercise: Check license state

```
SHOW LICENSE;  
SHOW STATS;  
SHOW WAL;  
SHOW WORKERS;
```

Expected learning outcome

- You can explain what license diagnostics means in ordinary language.
- You can identify the exact command that creates, changes, or reads state.
- You can inspect the response and decide whether the operation succeeded.
- You can name one security or performance consideration before production use.

Practice notes

Repeat the example with one changed value. Then deliberately make a small mistake, such as using a missing table or an unauthorized user, and observe the error. Understanding failure responses is as important as understanding successful commands.

Record the command sequence in a team notebook or deployment guide. A short, tested sequence is more valuable than a long memory of how the system probably behaves.

Workbook 15. Inspect Router status

This workbook lesson turns the idea of gateway operations into a repeatable exercise. The goal is not to memorize a command, but to understand what changes in the database, what the response should look like, and what the operator should check before moving to a larger data set.

Begin by reading the example slowly. Identify the database, the table, the key columns, the command that changes state, and the command that verifies the result. This habit is useful for TigerDB, TLang applications, and Router deployments because all three systems reward clear state transitions.

When practicing gateway operations, use a clean database or a table name that will not conflict with production data. Run every command once, inspect the JSON response, and repeat the same operation through the driver or frontend program that will eventually use it.

Exercise: Inspect Router status

```
ROUTER STATUS;  
ROUTER BACKENDS;  
ROUTER DATABASES;  
ROUTER HEALTH STATUS;  
ROUTER CLIENTS;
```

Expected learning outcome

- You can explain what gateway operations means in ordinary language.
- You can identify the exact command that creates, changes, or reads state.
- You can inspect the response and decide whether the operation succeeded.
- You can name one security or performance consideration before production use.

Practice notes

Repeat the example with one changed value. Then deliberately make a small mistake, such as using a missing table or an unauthorized user, and observe the error. Understanding failure responses is as important as understanding successful commands.

Record the command sequence in a team notebook or deployment guide. A short, tested sequence is more valuable than a long memory of how the system probably behaves.

Workbook 16. Reload Router safely

This workbook lesson turns the idea of configuration operations into a repeatable exercise. The goal is not to memorize a command, but to understand what changes in the database, what the response should look like, and what the operator should check before moving to a larger data set.

Begin by reading the example slowly. Identify the database, the table, the key columns, the command that changes state, and the command that verifies the result. This habit is useful for TigerDB, TLang applications, and Router deployments because all three systems reward clear state transitions.

When practicing configuration operations, use a clean database or a table name that will not conflict with production data. Run every command once, inspect the JSON response, and repeat the same operation through the driver or frontend program that will eventually use it.

Exercise: Reload Router safely

```
-- After editing tigerdb_router.config
ROUTER RELOAD;
ROUTER DATABASES;
ROUTER BACKENDS;
```

Expected learning outcome

- You can explain what configuration operations means in ordinary language.
- You can identify the exact command that creates, changes, or reads state.
- You can inspect the response and decide whether the operation succeeded.
- You can name one security or performance consideration before production use.

Practice notes

Repeat the example with one changed value. Then deliberately make a small mistake, such as using a missing table or an unauthorized user, and observe the error. Understanding failure responses is as important as understanding successful commands.

Record the command sequence in a team notebook or deployment guide. A short, tested sequence is more valuable than a long memory of how the system probably behaves.

Extended TigerDB SQL Cookbook

This cookbook collects additional TigerDB command patterns. Each recipe shows the intent, the SQL, and a verification habit. Copy a recipe into a test database before adapting it to a production schema.

Recipe 1: Bulk load with multi-row INSERT

Use this recipe when the application needs this exact pattern or when a developer wants to test a similar behavior with a smaller sample table. After running the SQL, inspect the JSON response and compare it with the expected row count or result shape.

```
INSERT INTO sales VALUES
(1, 101, 750.25, '2026-04'),
(2, 102, 1250.00, '2026-04'),
(3, 101, 300.50, '2026-05');
SELECT COUNT(*) FROM sales;
```

Verification habit: run a SELECT or SHOW command immediately after the operation. For writes, verify the affected row. For reports, verify a simple count or total by hand on a small input set.

Recipe 2: Monthly total report

Use this recipe when the application needs this exact pattern or when a developer wants to test a similar behavior with a smaller sample table. After running the SQL, inspect the JSON response and compare it with the expected row count or result shape.

```
SELECT month, COUNT(*), SUM(amount), AVG(amount)
FROM sales
GROUP BY month
ORDER BY month;
```

Verification habit: run a SELECT or SHOW command immediately after the operation. For writes, verify the affected row. For reports, verify a simple count or total by hand on a small input set.

Recipe 3: Customer sales join

Use this recipe when the application needs this exact pattern or when a developer wants to test a similar behavior with a smaller sample table. After running the SQL, inspect the JSON response and compare it with the expected row count or result shape.

```
SELECT customers.name, sales.amount, sales.month
FROM customers JOIN sales ON customers.id = sales.customer_id
ORDER BY sales.amount DESC;
```

Verification habit: run a SELECT or SHOW command immediately after the operation. For writes, verify the affected row. For reports, verify a simple count or total by hand on a small input set.

Recipe 4: Left join for missing related records

Use this recipe when the application needs this exact pattern or when a developer wants to test a similar behavior with a smaller sample table. After running the SQL, inspect the JSON response and compare it with the expected row count or result shape.

```
SELECT customers.id, customers.name, sales.amount
FROM customers LEFT OUTER JOIN sales ON customers.id = sales.customer_id;
```

Verification habit: run a SELECT or SHOW command immediately after the operation. For writes, verify the affected row. For reports, verify a simple count or total by hand on a small input set.

Recipe 5: Statistical report

Use this recipe when the application needs this exact pattern or when a developer wants to test a similar behavior with a smaller sample table. After running the SQL, inspect the JSON response and compare it with the expected row count or result shape.

```
SELECT COUNT(*), AVG(amount), STDDEV_POP(amount), MIN(amount), MAX(amount)
FROM sales;
```

Verification habit: run a SELECT or SHOW command immediately after the operation. For writes, verify the affected row. For reports, verify a simple count or total by hand on a small input set.

Recipe 6: Rounding and scalar functions

Use this recipe when the application needs this exact pattern or when a developer wants to test a similar behavior with a smaller sample table. After running the SQL, inspect the JSON response and compare it with the expected row count or result shape.

```
SELECT id, ROUND(amount), FLOOR(amount), CEIL(amount), ABS(amount)
FROM sales;
```

Verification habit: run a SELECT or SHOW command immediately after the operation. For writes, verify the affected row. For reports, verify a simple count or total by hand on a small input set.

Recipe 7: Power and logarithm functions

Use this recipe when the application needs this exact pattern or when a developer wants to test a similar behavior with a smaller sample table. After running the SQL, inspect the JSON response and compare it with the expected row count or result shape.

```
SELECT id, SQRT(amount), LOG(amount), LOG10(amount), POW(amount, 2)
FROM sales;
```

Verification habit: run a SELECT or SHOW command immediately after the operation. For writes, verify the affected row. For reports, verify a simple count or total by hand on a small input set.

Recipe 8: Market indicator query

Use this recipe when the application needs this exact pattern or when a developer wants to test a similar behavior with a smaller sample table. After running the SQL, inspect the JSON response and compare it with the expected row count or result shape.

```
SELECT ts, close, SMA(close,20), EMA(close,20), RSI(close,14)
FROM ticks
ORDER BY ts;
```

Verification habit: run a SELECT or SHOW command immediately after the operation. For writes, verify the affected row. For reports, verify a simple count or total by hand on a small input set.

Recipe 9: Procedure returns a report

Use this recipe when the application needs this exact pattern or when a developer wants to test a similar behavior with a smaller sample table. After running the SQL, inspect the JSON response and compare it with the expected row count or result shape.

```
CREATE PROCEDURE monthly_sales ON sales (IN p_month STRING) AS $$
SELECT customer_id, SUM(amount) FROM sales WHERE month = :p_month GROUP BY customer_id
$$;
CALL monthly_sales('2026-04');
```

Verification habit: run a SELECT or SHOW command immediately after the operation. For writes, verify the affected row. For reports, verify a simple count or total by hand on a small input set.

Recipe 10: Procedure inserts and returns row

Use this recipe when the application needs this exact pattern or when a developer wants to test a similar behavior with a smaller sample table. After running the SQL, inspect the JSON response and compare it with the expected row count or result shape.

```
CREATE PROCEDURE add_sale ON sales
(IN p_id INT, IN p_customer INT, IN p_amount DOUBLE, IN p_month STRING) AS $$
INSERT INTO sales VALUES (:p_id, :p_customer, :p_amount, :p_month);
SELECT * FROM sales WHERE id = :p_id
$$;
```

Verification habit: run a SELECT or SHOW command immediately after the operation. For writes, verify the affected row. For reports, verify a simple count or total by hand on a small input set.

Recipe 11: Procedure with ELSEIF

Use this recipe when the application needs this exact pattern or when a developer wants to test a similar behavior with a smaller sample table. After running the SQL, inspect the JSON response and compare it with the expected row count or result shape.

```
CREATE PROCEDURE bucket_sale ON sales (IN p_id INT, IN p_amount DOUBLE) AS $$
DECLARE v_bucket STRING DEFAULT 'small';
IF :p_amount >= 1000 THEN
SET v_bucket = 'large';
ELSEIF :p_amount >= 500 THEN
SET v_bucket = 'medium';
ELSE
SET v_bucket = 'small';
ENDIF;
UPDATE sales SET bucket = :v_bucket WHERE id = :p_id;
SELECT id, bucket FROM sales WHERE id = :p_id
$$;
```

Verification habit: run a SELECT or SHOW command immediately after the operation. For writes, verify the affected row. For reports, verify a simple count or total by hand on a small input set.

Recipe 12: Procedure tamper check habit

Use this recipe when the application needs this exact pattern or when a developer wants to test a similar behavior with a smaller sample table. After running the SQL, inspect the JSON response and compare it with the expected row count or result shape.

```
SHOW PROCEDURES ON sales;
-- valid=false or HASH_MISMATCH means the .sp file was changed outside TigerDB.
-- Recreate the procedure through CREATE PROCEDURE rather than executing a tampered file.
```

Verification habit: run a SELECT or SHOW command immediately after the operation. For writes, verify the affected row. For reports, verify a simple count or total by hand on a small input set.

Recipe 13: Read-only reporting user test

Use this recipe when the application needs this exact pattern or when a developer wants to test a similar behavior with a smaller sample table. After running the SQL, inspect the JSON response and compare it with the expected row count or result shape.

```
LOGIN report_user IDENTIFIED BY 'reportpass';
USE sales;
SELECT COUNT(*) FROM sales;
-- INSERT should fail unless the user has WRITE or FULL.
```

Verification habit: run a SELECT or SHOW command immediately after the operation. For writes, verify the affected row. For reports, verify a simple count or total by hand on a small input set.

Recipe 14: Admin diagnostics

Use this recipe when the application needs this exact pattern or when a developer wants to test a similar behavior with a smaller sample table. After running the SQL, inspect the JSON response and compare it with the expected row count or result shape.

```
SHOW LICENSE;  
SHOW PERFORMANCE;  
SHOW MEMORY;  
SHOW WAL;  
SHOW WORKERS;  
SHOW STATS;
```

Verification habit: run a SELECT or SHOW command immediately after the operation. For writes, verify the affected row. For reports, verify a simple count or total by hand on a small input set.

Recipe 15: Large batch benchmark shape

Use this recipe when the application needs this exact pattern or when a developer wants to test a similar behavior with a smaller sample table. After running the SQL, inspect the JSON response and compare it with the expected row count or result shape.

```
python3 benchmark_insert.py --rows 100000 --batch-size 10000  
python3 benchmark_select.py --rows 100000 --mode fullscan --runs 3
```

Verification habit: run a SELECT or SHOW command immediately after the operation. For writes, verify the affected row. For reports, verify a simple count or total by hand on a small input set.

Extended TLang Cookbook

These examples focus on frontend programming patterns. They assume TLang Phase 4.4 and show how to combine screens, environment passwords, RecordSets, procedure calls, and basic control flow.

TLang example 1: Environment password connection

This pattern is intended to be short enough to paste into a larger program. Keep the database work, screen rendering, and input handling separated so the generated application remains maintainable.

```
CONNECT TO TIGERDB SERVER HOST "127.0.0.1" ON PORT 9191
USER "admin" PASSWORD ENV "TIGERDB_PASSWORD"
DATABASE "sales" AS PD
```

- Compile the program after adding this block.
- Run with a test database first.
- Use PASSWORD ENV for real credentials.
- Close the connection before exit.

TLang example 2: Router connection

This pattern is intended to be short enough to paste into a larger program. Keep the database work, screen rendering, and input handling separated so the generated application remains maintainable.

```
CONNECT TO TIGERDB ROUTER HOST "127.0.0.1" ON PORT 9292
USER "admin" PASSWORD ENV "TIGERDB_PASSWORD"
DATABASE "sales" AS PD
```

- Compile the program after adding this block.
- Run with a test database first.
- Use PASSWORD ENV for real credentials.
- Close the connection before exit.

TLang example 3: Raw query error check

This pattern is intended to be short enough to paste into a larger program. Keep the database work, screen rendering, and input handling separated so the generated application remains maintainable.

```
var raw = PD.query("SELECT * FROM customers")
IF OK(raw) THEN
  @3,1 SAY "Query succeeded"
ELSE
  @3,1 SAY ERROR(raw)
ENDIF
```

- Compile the program after adding this block.
- Run with a test database first.
- Use PASSWORD ENV for real credentials.

- Close the connection before exit.

TLang example 4: RecordSet query

This pattern is intended to be short enough to paste into a larger program. Keep the database work, screen rendering, and input handling separated so the generated application remains maintainable.

```
RecordSet rs = PD.queryRS("SELECT id, name FROM customers")
FOREACH row IN rs
    @rownum,1 SAY row["id"]
    @rownum,10 SAY row["name"]
    rownum = rownum + 1
ENDFOREACH
```

- Compile the program after adding this block.
- Run with a test database first.
- Use PASSWORD ENV for real credentials.
- Close the connection before exit.

TLang example 5: Procedure call with status

This pattern is intended to be short enough to paste into a larger program. Keep the database work, screen rendering, and input handling separated so the generated application remains maintainable.

```
var result = PD.call("record_sale", 1, 101, 750.25)
IF OK(result) THEN
    @10,1 SAY "Recorded"
ELSE
    @10,1 SAY ERROR(result)
ENDIF
```

- Compile the program after adding this block.
- Run with a test database first.
- Use PASSWORD ENV for real credentials.
- Close the connection before exit.

TLang example 6: Procedure call returning rows

This pattern is intended to be short enough to paste into a larger program. Keep the database work, screen rendering, and input handling separated so the generated application remains maintainable.

```
RecordSet rs = PD.callRS("monthly_sales", "2026-04")
FOREACH row IN rs
    @rownum,1 SAY row["customer_id"]
    @rownum,20 SAY row["SUM"]
    rownum = rownum + 1
ENDFOREACH
```

- Compile the program after adding this block.

- Run with a test database first.
- Use PASSWORD ENV for real credentials.
- Close the connection before exit.

TLang example 7: Simple menu shell

This pattern is intended to be short enough to paste into a larger program. Keep the database work, screen rendering, and input handling separated so the generated application remains maintainable.

```
DEFINE MENU main
ADD ITEM "list" PROMPT "List customers"
ADD ITEM "quit" PROMPT "Quit"
-- Callback wiring depends on the application structure.
```

- Compile the program after adding this block.
- Run with a test database first.
- Use PASSWORD ENV for real credentials.
- Close the connection before exit.

TLang example 8: Screen table output

This pattern is intended to be short enough to paste into a larger program. Keep the database work, screen rendering, and input handling separated so the generated application remains maintainable.

```
DEFINE TABLE FROM 2,1 TO 12,78 DOUBLE
@3,3 SAY "ID"
@3,12 SAY "Name"
@3,45 SAY "Amount"
```

- Compile the program after adding this block.
- Run with a test database first.
- Use PASSWORD ENV for real credentials.
- Close the connection before exit.

TLang example 9: Input form pattern

This pattern is intended to be short enough to paste into a larger program. Keep the database work, screen rendering, and input handling separated so the generated application remains maintainable.

```
@5,3 SAY "Customer ID:"
@5,18 GET p_customer_id
@6,3 SAY "Amount:"
@6,18 GET p_amount
READ
```

- Compile the program after adding this block.
- Run with a test database first.
- Use PASSWORD ENV for real credentials.
- Close the connection before exit.

TLang example 10: Controlled loop

This pattern is intended to be short enough to paste into a larger program. Keep the database work, screen rendering, and input handling separated so the generated application remains maintainable.

```
var i = 1
WHILE i <= 10 DO
  @i,1 SAY i
  i = i + 1
ENDWHILE
```

- Compile the program after adding this block.
- Run with a test database first.
- Use PASSWORD ENV for real credentials.
- Close the connection before exit.

TLang example 11: Branching by result

This pattern is intended to be short enough to paste into a larger program. Keep the database work, screen rendering, and input handling separated so the generated application remains maintainable.

```
IF amount >= 1000 THEN
  bucket = "large"
ELSEIF amount >= 500 THEN
  bucket = "medium"
ELSE
  bucket = "small"
ENDIF
```

- Compile the program after adding this block.
- Run with a test database first.
- Use PASSWORD ENV for real credentials.
- Close the connection before exit.

TLang example 12: Close connection

This pattern is intended to be short enough to paste into a larger program. Keep the database work, screen rendering, and input handling separated so the generated application remains maintainable.

```
PD.close()
@23,1 SAY "Session closed"
```

- Compile the program after adding this block.
- Run with a test database first.
- Use PASSWORD ENV for real credentials.
- Close the connection before exit.

TLang example 13: Environment helper

This pattern is intended to be short enough to paste into a larger program. Keep the database work, screen rendering, and input handling separated so the generated application remains maintainable.

```
var server = ENV("TIGERDB_HOST")
IF server = "" THEN
  server = "127.0.0.1"
ENDIF
```

- Compile the program after adding this block.
- Run with a test database first.
- Use PASSWORD ENV for real credentials.
- Close the connection before exit.

TLang example 14: Convert raw JSON to RecordSet

This pattern is intended to be short enough to paste into a larger program. Keep the database work, screen rendering, and input handling separated so the generated application remains maintainable.

```
var raw = PD.query("SELECT * FROM customers")
RecordSet rs = TO_RS(raw)
```

- Compile the program after adding this block.
- Run with a test database first.
- Use PASSWORD ENV for real credentials.
- Close the connection before exit.

TLang example 15: Reusable display function

This pattern is intended to be short enough to paste into a larger program. Keep the database work, screen rendering, and input handling separated so the generated application remains maintainable.

```
FUNCTION show_customer(row)
  @rownum,1 SAY row["id"]
  @rownum,10 SAY row["name"]
ENDFUNCTION
```

- Compile the program after adding this block.
- Run with a test database first.
- Use PASSWORD ENV for real credentials.
- Close the connection before exit.

Extended Router Operations Playbook

Router operations should be repeatable. Each playbook below explains a small operational task that an administrator can run and verify without reading the source code.

Router playbook 1: Check Router health

Use `ROUTER HEALTH` for a live check and `ROUTER HEALTH STATUS` for cached status.

```
ROUTER HEALTH;  
ROUTER HEALTH STATUS;
```

Verification habit: confirm the JSON response says `ok:true` or that the status fields match the expected route, backend, or session state. If a change affects credentials, test with a non-admin user before considering the change complete.

Router playbook 2: List database routes

Confirm that every database points to the expected backend.

```
ROUTER DATABASES;
```

Verification habit: confirm the JSON response says `ok:true` or that the status fields match the expected route, backend, or session state. If a change affects credentials, test with a non-admin user before considering the change complete.

Router playbook 3: Inspect pools

Pool output helps identify idle backend sessions and reuse behavior.

```
ROUTER POOLS;
```

Verification habit: confirm the JSON response says `ok:true` or that the status fields match the expected route, backend, or session state. If a change affects credentials, test with a non-admin user before considering the change complete.

Router playbook 4: Inspect active clients

`ROUTER CLIENTS` shows safe metadata without passwords.

```
ROUTER CLIENTS;
```

Verification habit: confirm the JSON response says `ok:true` or that the status fields match the expected route, backend, or session state. If a change affects credentials, test with a non-admin user before considering the change complete.

Router playbook 5: Reload after route edit

Use reload after changing backend, database, user, grant, or policy lines.

```
ROUTER RELOAD;  
ROUTER DATABASES;  
ROUTER BACKENDS;
```

Verification habit: confirm the JSON response says ok:true or that the status fields match the expected route, backend, or session state. If a change affects credentials, test with a non-admin user before considering the change complete.

Router playbook 6: Add a reporting user

A reporting user should have READ but not WRITE or FULL.

```
USER report reportpass report reportpass
GRANT report sales READ
ROUTER RELOAD;
```

Verification habit: confirm the JSON response says ok:true or that the status fields match the expected route, backend, or session state. If a change affects credentials, test with a non-admin user before considering the change complete.

Router playbook 7: Add an ingest user

An ingest user can write data but should not manage procedures or routes.

```
USER ingest ingestpass ingest ingestpass
GRANT ingest sales WRITE
ROUTER RELOAD;
```

Verification habit: confirm the JSON response says ok:true or that the status fields match the expected route, backend, or session state. If a change affects credentials, test with a non-admin user before considering the change complete.

Router playbook 8: Conservative CALL policy

Require FULL for CALL when backend mappings are not one-to-one.

```
STORED_PROCEDURE_CALL_ROLE=FULL
STORED_PROCEDURE_CALL_REQUIRE_SAME_BACKEND_USER=ON
```

Verification habit: confirm the JSON response says ok:true or that the status fields match the expected route, backend, or session state. If a change affects credentials, test with a non-admin user before considering the change complete.

Router playbook 9: Invoker-safe CALL policy

Allow READ only when Router and backend users match.

```
STORED_PROCEDURE_CALL_ROLE=READ
STORED_PROCEDURE_CALL_REQUIRE_SAME_BACKEND_USER=ON
```

Verification habit: confirm the JSON response says ok:true or that the status fields match the expected route, backend, or session state. If a change affects credentials, test with a non-admin user before considering the change complete.

Router playbook 10: Throttle login failures

Enable auth throttling to reduce brute-force pressure.

```

AUTH_THROTTLE=ON
AUTH_FAIL_WINDOW_SECONDS=60
AUTH_FAIL_THRESHOLD=5
AUTH_THROTTLE_SECONDS=5

```

Verification habit: confirm the JSON response says ok:true or that the status fields match the expected route, backend, or session state. If a change affects credentials, test with a non-admin user before considering the change complete.

Router playbook 11: Backend migration

Move a database by changing its DATABASE route and reloading.

```

BACKEND main 10.0.0.10 9191
BACKEND newmain 10.0.0.11 9191
DATABASE sales newmain
ROUTER RELOAD;

```

Verification habit: confirm the JSON response says ok:true or that the status fields match the expected route, backend, or session state. If a change affects credentials, test with a non-admin user before considering the change complete.

Router playbook 12: Private backend deployment

Expose Router to applications, keep backend ports private.

```

# Firewall idea:
# allow application_net -> router:9292
# allow router_host -> backend:9191
# deny public -> backend:9191

```

Verification habit: confirm the JSON response says ok:true or that the status fields match the expected route, backend, or session state. If a change affects credentials, test with a non-admin user before considering the change complete.

Router playbook 13: Large SQL through Router

Use the R2+ Router line so large batch INSERT SQL is forwarded without an 8192-byte extraction limit.

```
python3 benchmark_insert.py --host 127.0.0.1 --port 9292 --rows 100000 --batch-size 10000
```

Verification habit: confirm the JSON response says ok:true or that the status fields match the expected route, backend, or session state. If a change affects credentials, test with a non-admin user before considering the change complete.

Router playbook 14: Session diagnosis

Find idle or unexpected clients before restarting.

```

ROUTER CLIENTS;
ROUTER STATUS;

```

Verification habit: confirm the JSON response says ok:true or that the status fields match the expected route, backend, or session state. If a change affects credentials, test with a non-admin user before considering the change complete.

Router playbook 15: Graceful operational check

After a restart, run these commands before releasing traffic.

```
ROUTER STATUS;  
ROUTER BACKENDS;  
ROUTER HEALTH;  
ROUTER DATABASES;
```

Verification habit: confirm the JSON response says ok:true or that the status fields match the expected route, backend, or session state. If a change affects credentials, test with a non-admin user before considering the change complete.

Failure Modes and Recovery Narratives

Scenario 1: License missing at startup

The server allows diagnostic checks but blocks normal database work. Run `SHOW LICENSE`, check `LICENSE_KEY`, activate the server, then retry the application command.

- State what command failed.
- Check the nearest diagnostic command.
- Confirm whether the error occurs direct to TigerDB and through Router.
- Make one change at a time and rerun the smallest possible test.

Baseline diagnostics

```
PING;
SHOW LICENSE;
SHOW STATS;
-- Router path:
ROUTER STATUS;
ROUTER HEALTH STATUS;
```

Scenario 2: Must change password

The user can login but cannot continue normal work until `CHANGE PASSWORD` succeeds. A client should detect `must_change_password` and prompt immediately.

- State what command failed.
- Check the nearest diagnostic command.
- Confirm whether the error occurs direct to TigerDB and through Router.
- Make one change at a time and rerun the smallest possible test.

Baseline diagnostics

```
PING;
SHOW LICENSE;
SHOW STATS;
-- Router path:
ROUTER STATUS;
ROUTER HEALTH STATUS;
```

Scenario 3: Procedure hash mismatch

`SHOW PROCEDURES` reports invalid status. The `.sp` file was modified outside TigerDB or restored incorrectly. Recreate the procedure through `CREATE PROCEDURE` or restore the correct file and catalog together.

- State what command failed.
- Check the nearest diagnostic command.
- Confirm whether the error occurs direct to TigerDB and through Router.
- Make one change at a time and rerun the smallest possible test.

Baseline diagnostics

```

PING;
SHOW LICENSE;
SHOW STATS;
-- Router path:
ROUTER STATUS;
ROUTER HEALTH STATUS;

```

Scenario 4: Large batch insert through old Router

The direct server accepts the large INSERT but Router fails. Use Router R2 or later so the sql field is extracted dynamically.

- State what command failed.
- Check the nearest diagnostic command.
- Confirm whether the error occurs direct to TigerDB and through Router.
- Make one change at a time and rerun the smallest possible test.

Baseline diagnostics

```

PING;
SHOW LICENSE;
SHOW STATS;
-- Router path:
ROUTER STATUS;
ROUTER HEALTH STATUS;

```

Scenario 5: Slow indexed inserts

Unindexed insert remains fast but indexed insert drops sharply at scale. Measure secondary-index maintenance separately and optimize in an experimental branch only.

- State what command failed.
- Check the nearest diagnostic command.
- Confirm whether the error occurs direct to TigerDB and through Router.
- Make one change at a time and rerun the smallest possible test.

Baseline diagnostics

```

PING;
SHOW LICENSE;
SHOW STATS;
-- Router path:
ROUTER STATUS;
ROUTER HEALTH STATUS;

```

Scenario 6: Router user removed during reload

An active session may remain connected until the next request, but R8 session snapshots cause later requests to refresh the user record and deny removed users.

- State what command failed.
- Check the nearest diagnostic command.
- Confirm whether the error occurs direct to TigerDB and through Router.

- Make one change at a time and rerun the smallest possible test.

Baseline diagnostics

```
PING;
SHOW LICENSE;
SHOW STATS;
-- Router path:
ROUTER STATUS;
ROUTER HEALTH STATUS;
```

Scenario 7: Backend unhealthy

ROUTER DATABASES may show backend_unhealthy. Run ROUTER HEALTH, inspect last_error, and test direct backend PING from the Router host.

- State what command failed.
- Check the nearest diagnostic command.
- Confirm whether the error occurs direct to TigerDB and through Router.
- Make one change at a time and rerun the smallest possible test.

Baseline diagnostics

```
PING;
SHOW LICENSE;
SHOW STATS;
-- Router path:
ROUTER STATUS;
ROUTER HEALTH STATUS;
```

Scenario 8: Client session idle

ROUTER CLIENTS shows long idle seconds. Close stale clients or tune application connection usage.

- State what command failed.
- Check the nearest diagnostic command.
- Confirm whether the error occurs direct to TigerDB and through Router.
- Make one change at a time and rerun the smallest possible test.

Baseline diagnostics

```
PING;
SHOW LICENSE;
SHOW STATS;
-- Router path:
ROUTER STATUS;
ROUTER HEALTH STATUS;
```

Scenario 9: ORDER BY slower than full scan

Sorting costs more than scanning. Confirm whether the application actually needs ordered rows and use LIMIT where practical.

- State what command failed.

- Check the nearest diagnostic command.
- Confirm whether the error occurs direct to TigerDB and through Router.
- Make one change at a time and rerun the smallest possible test.

Baseline diagnostics

```
PING;
SHOW LICENSE;
SHOW STATS;
-- Router path:
ROUTER STATUS;
ROUTER HEALTH STATUS;
```

Scenario 10: Configuration typo

ROUTER RELOAD should reject invalid configuration without replacing the running config. Fix the file and reload again.

- State what command failed.
- Check the nearest diagnostic command.
- Confirm whether the error occurs direct to TigerDB and through Router.
- Make one change at a time and rerun the smallest possible test.

Baseline diagnostics

```
PING;
SHOW LICENSE;
SHOW STATS;
-- Router path:
ROUTER STATUS;
ROUTER HEALTH STATUS;
```

Scenario 11: Benchmark timeout during login

The failure happened before data loading. Test PING and LOGIN directly, confirm the server is responsive, and restart if a previous benchmark left the process busy.

- State what command failed.
- Check the nearest diagnostic command.
- Confirm whether the error occurs direct to TigerDB and through Router.
- Make one change at a time and rerun the smallest possible test.

Baseline diagnostics

```
PING;
SHOW LICENSE;
SHOW STATS;
-- Router path:
ROUTER STATUS;
ROUTER HEALTH STATUS;
```

Scenario 12: Permission denied from Router

Check both Router GRANT and backend TigerDB role. A Router grant cannot override missing backend permissions.

- State what command failed.
- Check the nearest diagnostic command.
- Confirm whether the error occurs direct to TigerDB and through Router.
- Make one change at a time and rerun the smallest possible test.

Baseline diagnostics

```
PING;  
SHOW LICENSE;  
SHOW STATS;  
-- Router path:  
ROUTER STATUS;  
ROUTER HEALTH STATUS;
```

Final Deployment Checklist

Use this checklist before presenting a TigerDB deployment to users or connecting a TLang application to production data. It combines the server, Router, TLang, drivers, configuration, and operational checks covered throughout the book.

TigerDB Server

- Optimized binary built from stable R4c baseline
- tigerdb.config reviewed and protected
- LICENSE_KEY configured and SHOW LICENSE passes
- Default admin password changed
- Application users created with least privilege
- Small create/insert/select smoke test passes
- Stored procedures listed and valid
- Benchmark matrix recorded for current hardware

TLang

- Phase 4.4 compiler builds successfully
- Programs use PASSWORD ENV for real credentials
- RecordSet loops tested with small result sets
- PD.call and PD.callRS tested against procedures
- Error handling uses OK and ERROR where raw JSON is used
- Generated C application compiles and runs on the target host

Router

- Router R8 binary installed
- tigerdb_router.config protected by OS permissions
- Backends reachable only from approved hosts
- ROUTER HEALTH and ROUTER DATABASES pass
- ROUTER CLIENTS shows expected sessions only
- ROUTER RELOAD tested after config edit
- AUTH_THROTTLE enabled
- CALL policy matches backend identity model

Operations

- Backups include data directories and relevant configs
- Firewall rules documented
- Benchmark scripts kept with results
- Troubleshooting playbook available
- All credentials rotated from defaults
- Users know whether to connect direct or through Router

TigerDB Server Command Atlas

The following pages provide one-page entries for TigerDB Server commands, functions, and procedure-language statements. This atlas is intentionally redundant with earlier chapters so that readers can find a command quickly without re-reading the tutorial.

TigerDB Command 1. PING

Field	Value
Category	Diagnostics
Purpose	Check that the server protocol is responsive.
Related	SHOW LICENSE

PING belongs to the Diagnostics group. Its purpose is to check that the server protocol is responsive. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
PING;
```

Example

```
PING;
```

Operational notes

- Allowed as a diagnostic command.
- Use before login when testing reachability.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 2. LOGIN

Field	Value
Category	Authentication
Purpose	Authenticate a TigerDB user.
Related	CHANGE PASSWORD

LOGIN belongs to the Authentication group. Its purpose is to authenticate a tigerdb user. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
LOGIN user IDENTIFIED BY 'password';
```

Example

```
LOGIN admin IDENTIFIED BY 'admin1234';
```

Operational notes

- Required before normal database work.
- A first-time user may be required to change password.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 3. CHANGE PASSWORD

Field	Value
Category	Authentication
Purpose	Change the current user password.
Related	LOGIN

CHANGE PASSWORD belongs to the Authentication group. Its purpose is to change the current user password. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
CHANGE PASSWORD TO 'new_password';
```

Example

```
CHANGE PASSWORD TO 'better_password';
```

Operational notes

- Do not put password changes inside stored procedures.
- Restart clients that cache credentials.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 4. SHOW LICENSE

Field	Value
Category	Diagnostics
Purpose	Inspect the license state.
Related	PING

SHOW LICENSE belongs to the Diagnostics group. Its purpose is to inspect the license state. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SHOW LICENSE;
```

Example

```
SHOW LICENSE;
```

Operational notes

- Useful when normal commands are blocked.
- A missing or invalid license blocks protected actions.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 5. SHOW STATS

Field	Value
Category	Diagnostics
Purpose	Inspect request and write counters.
Related	SHOW PERFORMANCE

SHOW STATS belongs to the Diagnostics group. Its purpose is to inspect request and write counters. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SHOW STATS;
```

Example

```
SHOW STATS;
```

Operational notes

- Useful before and after benchmarks.
- Counters help confirm expected row activity.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 6. SHOW PERFORMANCE

Field	Value
Category	Diagnostics
Purpose	Inspect performance-oriented counters.
Related	SHOW STATS

SHOW PERFORMANCE belongs to the Diagnostics group. Its purpose is to inspect performance-oriented counters. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SHOW PERFORMANCE;
```

Example

```
SHOW PERFORMANCE;
```

Operational notes

- Use together with benchmark scripts.
- Do not compare debug and optimized builds.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 7. SHOW MEMORY

Field	Value
Category	Diagnostics
Purpose	Inspect memory-related status.
Related	SHOW STATS

SHOW MEMORY belongs to the Diagnostics group. Its purpose is to inspect memory-related status. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SHOW MEMORY;
```

Example

```
SHOW MEMORY;
```

Operational notes

- Useful before large loads.
- Watch memory pressure during batch tests.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 8. SHOW WAL

Field	Value
Category	Diagnostics
Purpose	Inspect write-ahead log status.
Related	SHOW STATS

SHOW WAL belongs to the Diagnostics group. Its purpose is to inspect write-ahead log status. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SHOW WAL;
```

Example

```
SHOW WAL;
```

Operational notes

- WAL counters should match write workload.
- Large WAL growth may trigger maintenance.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 9. SHOW WORKERS

Field	Value
Category	Diagnostics
Purpose	Inspect worker configuration.
Related	tigerdb.config

SHOW WORKERS belongs to the Diagnostics group. Its purpose is to inspect worker configuration. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SHOW WORKERS;
```

Example

```
SHOW WORKERS;
```

Operational notes

- Confirms read, write, and connection worker counts.
- AUTO_TUNE may choose defaults.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 10. CREATE DATABASE

Field	Value
Category	Data definition
Purpose	Create a physical database.
Related	USE

CREATE DATABASE belongs to the Data definition group. Its purpose is to create a physical database. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
CREATE DATABASE database_name;
```

Example

```
CREATE DATABASE sales;
```

Operational notes

- Requires administrative or FULL permission.
- Use business-domain names.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 11. SHOW DATABASES

Field	Value
Category	Metadata
Purpose	List available databases.
Related	CREATE DATABASE

SHOW DATABASES belongs to the Metadata group. Its purpose is to list available databases. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SHOW DATABASES;
```

Example

```
SHOW DATABASES;
```

Operational notes

- Useful after CREATE DATABASE.
- Through Router, routes may also affect visibility.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 12. USE

Field	Value
Category	Session context
Purpose	Select the current database.
Related	SHOW TABLES

USE belongs to the Session context group. Its purpose is to select the current database. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
USE database_name;
```

Example

```
USE sales;
```

Operational notes

- Run before table commands.
- Router maps the database name to a backend.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 13. CREATE TABLE

Field	Value
Category	Data definition
Purpose	Create a table schema.
Related	SHOW TABLE

CREATE TABLE belongs to the Data definition group. Its purpose is to create a table schema. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
CREATE TABLE table_name (id INT PRIMARY KEY, name STRING);
```

Example

```
CREATE TABLE customers (id INT PRIMARY KEY, name STRING, amount DOUBLE);
```

Operational notes

- Pick the primary key carefully.
- Keep column names meaningful.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 14. SHOW TABLES

Field	Value
Category	Metadata
Purpose	List tables in the current database.
Related	SHOW TABLE

SHOW TABLES belongs to the Metadata group. Its purpose is to list tables in the current database. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SHOW TABLES;
```

Example

```
SHOW TABLES;
```

Operational notes

- Use after USE database.
- Confirms table creation.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 15. SHOW TABLE

Field	Value
Category	Metadata
Purpose	Describe one table.
Related	CREATE TABLE

SHOW TABLE belongs to the Metadata group. Its purpose is to describe one table. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SHOW TABLE table_name;
```

Example

```
SHOW TABLE customers;
```

Operational notes

- Use before writing application code.
- Confirms columns and types.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 16. INSERT

Field	Value
Category	Data modification
Purpose	Add one or more records.
Related	SELECT

INSERT belongs to the Data modification group. Its purpose is to add one or more records. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
INSERT INTO table VALUES (...);
```

Example

```
INSERT INTO customers VALUES (1, 'Anil', 250.75);
```

Operational notes

- Batch inserts improve throughput.
- Primary keys must remain unique.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 17. BATCH INSERT

Field	Value
Category	Data modification
Purpose	Load many rows in one INSERT.
Related	SHOW WAL

BATCH INSERT belongs to the Data modification group. Its purpose is to load many rows in one insert. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
INSERT INTO table VALUES (...),(...),(...);
```

Example

```
INSERT INTO customers VALUES (1,'A',10.0),(2,'B',20.0);
```

Operational notes

- Keep request below protocol limit.
- Benchmark batch sizes.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 18. UPDATE

Field	Value
Category	Data modification
Purpose	Modify matching records.
Related	SELECT

UPDATE belongs to the Data modification group. Its purpose is to modify matching records. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
UPDATE table SET col = value WHERE key = value;
```

Example

```
UPDATE customers SET amount = 500.0 WHERE id = 1;
```

Operational notes

- Test the WHERE clause with SELECT first.
- Requires write permission.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 19. DELETE

Field	Value
Category	Data modification
Purpose	Delete matching records.
Related	SELECT

DELETE belongs to the Data modification group. Its purpose is to delete matching records. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
DELETE FROM table WHERE key = value;
```

Example

```
DELETE FROM customers WHERE id = 1;
```

Operational notes

- Prefer narrow WHERE clauses.
- Consider soft-delete columns for business data.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 20. SELECT

Field	Value
Category	Query
Purpose	Read records.
Related	WHERE

SELECT belongs to the Query group. Its purpose is to read records. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT columns FROM table WHERE condition;
```

Example

```
SELECT id, name FROM customers WHERE id = 1;
```

Operational notes

- Use projection instead of SELECT * when possible.
- Inspect JSON rows in driver code.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 21. WHERE

Field	Value
Category	Query
Purpose	Filter records.
Related	CREATE INDEX

WHERE belongs to the Query group. Its purpose is to filter records. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT * FROM table WHERE column = value;
```

Example

```
SELECT * FROM customers WHERE amount >= 100;
```

Operational notes

- Indexes help when filtering selective columns.
- Test with representative data sizes.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 22. JOIN

Field	Value
Category	Query
Purpose	Combine matching rows from two tables.
Related	LEFT OUTER JOIN

JOIN belongs to the Query group. Its purpose is to combine matching rows from two tables. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT ... FROM a JOIN b ON a.id = b.a_id;
```

Example

```
SELECT c.name, s.amount FROM customers c JOIN sales s ON c.id = s.customer_id;
```

Operational notes

- Use clear keys.
- Avoid ambiguous column names.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 23. LEFT OUTER JOIN

Field	Value
Category	Query
Purpose	Keep left rows even without right matches.
Related	JOIN

LEFT OUTER JOIN belongs to the Query group. Its purpose is to keep left rows even without right matches. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT ... FROM a LEFT OUTER JOIN b ON ...;
```

Example

```
SELECT c.name, s.amount FROM customers c LEFT OUTER JOIN sales s ON c.id=s.customer_id;
```

Operational notes

- Useful for missing-related-record reports.
- Expect null-like values for missing matches.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 24. GROUP BY

Field	Value
Category	Query
Purpose	Aggregate rows by category.
Related	COUNT

GROUP BY belongs to the Query group. Its purpose is to aggregate rows by category. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT category, COUNT(*) FROM table GROUP BY category;
```

Example

```
SELECT city, COUNT(*), SUM(amount) FROM customers GROUP BY city;
```

Operational notes

- Every selected non-aggregate should be grouped.
- Use small data to verify totals.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 25. ORDER BY

Field	Value
Category	Query
Purpose	Sort result rows.
Related	LIMIT

ORDER BY belongs to the Query group. Its purpose is to sort result rows. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT * FROM table ORDER BY column DESC;
```

Example

```
SELECT * FROM customers ORDER BY amount DESC LIMIT 10;
```

Operational notes

- Sorting costs more than scanning.
- Use LIMIT for screens.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 26. LIMIT

Field	Value
Category	Query
Purpose	Restrict rows returned.
Related	ORDER BY

LIMIT belongs to the Query group. Its purpose is to restrict rows returned. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT * FROM table LIMIT n;
```

Example

```
SELECT * FROM customers LIMIT 20;
```

Operational notes

- Good for UI pages.
- Combine with ORDER BY for top-N reports.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 27. CREATE INDEX

Field	Value
Category	Indexing
Purpose	Create a secondary index.
Related	SHOW INDEXES

CREATE INDEX belongs to the Indexing group. Its purpose is to create a secondary index. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
CREATE INDEX index_name ON table(column);
```

Example

```
CREATE INDEX idx_customers_city ON customers(city);
```

Operational notes

- Improves selective reads.
- Adds write maintenance cost.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 28. CREATE UNIQUE INDEX

Field	Value
Category	Indexing
Purpose	Create a unique-style index.
Related	CREATE INDEX

CREATE UNIQUE INDEX belongs to the Indexing group. Its purpose is to create a unique-style index. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
CREATE UNIQUE INDEX index_name ON table(column);
```

Example

```
CREATE UNIQUE INDEX idx_customers_name ON customers(name);
```

Operational notes

- Use only when uniqueness matters.
- Measure inserts after adding it.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 29. SHOW INDEXES

Field	Value
Category	Indexing
Purpose	List indexes on a table.
Related	CREATE INDEX

SHOW INDEXES belongs to the Indexing group. Its purpose is to list indexes on a table. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SHOW INDEXES ON table;
```

Example

```
SHOW INDEXES ON customers;
```

Operational notes

- Useful during performance diagnosis.
- Index names should be descriptive.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 30. COUNT

Field	Value
Category	Function
Purpose	Count rows.
Related	GROUP BY

COUNT belongs to the Function group. Its purpose is to count rows. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT COUNT(*) FROM table;
```

Example

```
SELECT COUNT(*) FROM customers;
```

Operational notes

- Good first verification after load.
- Can be used with GROUP BY.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 31. SUM

Field	Value
Category	Function
Purpose	Add numeric values.
Related	AVG

SUM belongs to the Function group. Its purpose is to add numeric values. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT SUM(column) FROM table;
```

Example

```
SELECT SUM(amount) FROM sales;
```

Operational notes

- Use with numeric columns.
- Check totals on small samples.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 32. AVG / MEAN

Field	Value
Category	Function
Purpose	Calculate average numeric value.
Related	SUM

AVG / MEAN belongs to the Function group. Its purpose is to calculate average numeric value. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT AVG(column) FROM table;
```

Example

```
SELECT AVG(amount) FROM sales;
```

Operational notes

- Mean and average are aliases in many reporting contexts.
- Be clear about filtered row set.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 33. MIN / MAX

Field	Value
Category	Function
Purpose	Find smallest or largest value.
Related	ORDER BY

MIN / MAX belongs to the Function group. Its purpose is to find smallest or largest value. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT MIN(column), MAX(column) FROM table;
```

Example

```
SELECT MIN(amount), MAX(amount) FROM sales;
```

Operational notes

- Useful for range checks.
- Combine with GROUP BY for category ranges.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 34. ABS

Field	Value
Category	Function
Purpose	Return absolute value.
Related	ROUND

ABS belongs to the Function group. Its purpose is to return absolute value. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT ABS(value) FROM table;
```

Example

```
SELECT ABS(amount) FROM sales;
```

Operational notes

- Scalar function.
- Works on numeric values.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 35. ROUND

Field	Value
Category	Function
Purpose	Round numeric values.
Related	FLOOR

ROUND belongs to the Function group. Its purpose is to round numeric values. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT ROUND(value) FROM table;
```

Example

```
SELECT ROUND(amount) FROM sales;
```

Operational notes

- Useful for display values.
- Keep raw values where precision matters.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 36. FLOOR

Field	Value
Category	Function
Purpose	Round down.
Related	CEIL

FLOOR belongs to the Function group. Its purpose is to round down. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT FLOOR(value) FROM table;
```

Example

```
SELECT FLOOR(amount) FROM sales;
```

Operational notes

- Scalar function.
- Negative values round toward lower integer.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 37. CEIL / CEILING

Field	Value
Category	Function
Purpose	Round up.
Related	FLOOR

CEIL / CEILING belongs to the Function group. Its purpose is to round up. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT CEIL(value) FROM table;
```

Example

```
SELECT CEIL(amount) FROM sales;
```

Operational notes

- Use CEIL or CEILING depending on code style.
- Useful for buckets and thresholds.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 38. SQRT

Field	Value
Category	Function
Purpose	Calculate square root.
Related	POW

SQRT belongs to the Function group. Its purpose is to calculate square root. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT SQRT(value) FROM table;
```

Example

```
SELECT SQRT(amount) FROM sales;
```

Operational notes

- Input should be non-negative.
- Part of math function set.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 39. LOG / LN / LOG10

Field	Value
Category	Function
Purpose	Calculate logarithms.
Related	EXP

LOG / LN / LOG10 belongs to the Function group. Its purpose is to calculate logarithms. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT LOG(value) FROM table;
```

Example

```
SELECT LOG10(amount) FROM sales;
```

Operational notes

- Input should be positive.
- Useful for analytical transformations.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 40. EXP

Field	Value
Category	Function
Purpose	Calculate exponential value.
Related	LOG

EXP belongs to the Function group. Its purpose is to calculate exponential value. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT EXP(value) FROM table;
```

Example

```
SELECT EXP(score) FROM metrics;
```

Operational notes

- Watch numeric growth.
- Pair with LOG in analysis.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 41. POW

Field	Value
Category	Function
Purpose	Raise value to a power.
Related	SQRT

POW belongs to the Function group. Its purpose is to raise value to a power. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT POW(value, exponent) FROM table;
```

Example

```
SELECT POW(amount, 2) FROM sales;
```

Operational notes

- Two-argument scalar function.
- Use with care for large values.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 42. STDDEV_POP

Field	Value
Category	Function
Purpose	Population standard deviation.
Related	STDDEV_SAMP

STDDEV_POP belongs to the Function group. Its purpose is to population standard deviation. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT STDDEV_POP(column) FROM table;
```

Example

```
SELECT STDDEV_POP(amount) FROM sales;
```

Operational notes

- Population statistic.
- Compare with sample statistic when needed.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 43. STDDEV_SAMP

Field	Value
Category	Function
Purpose	Sample standard deviation.
Related	STDDEV_POP

STDDEV_SAMP belongs to the Function group. Its purpose is to sample standard deviation. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT STDDEV_SAMP(column) FROM table;
```

Example

```
SELECT STDDEV_SAMP(amount) FROM sales;
```

Operational notes

- Sample statistic.
- Useful for sampled data sets.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 44. SMA / EMA / WMA

Field	Value
Category	Function
Purpose	Calculate moving averages.
Related	RSI

SMA / EMA / WMA belongs to the Function group. Its purpose is to calculate moving averages. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a `SHOW` or `SELECT` command.

Syntax

```
SELECT SMA(close,20) FROM ticks ORDER BY ts;
```

Example

```
SELECT ts, close, EMA(close,20) FROM ticks ORDER BY ts;
```

Operational notes

- Requires meaningful ORDER BY.
- Early rows may not have enough history.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 45. RSI

Field	Value
Category	Function
Purpose	Calculate relative strength index.
Related	SMA

RSI belongs to the Function group. Its purpose is to calculate relative strength index. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SELECT RSI(close,14) FROM ticks ORDER BY ts;
```

Example

```
SELECT ts, RSI(close,14) FROM ticks ORDER BY ts;
```

Operational notes

- Market-style ordered function.
- Verify on small known series.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 46. CREATE PROCEDURE

Field	Value
Category	Stored procedure
Purpose	Register a stored procedure.
Related	SHOW PROCEDURES

CREATE PROCEDURE belongs to the Stored procedure group. Its purpose is to register a stored procedure. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
CREATE PROCEDURE name ON table AS $$ SQL $$;
```

Example

```
CREATE PROCEDURE list_customers ON customers AS $$ SELECT * FROM customers $$;
```

Operational notes

- Creates .sp file and catalog metadata.
- Requires FULL permission.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 47. SHOW PROCEDURES

Field	Value
Category	Stored procedure
Purpose	List procedures and validity.
Related	CALL

SHOW PROCEDURES belongs to the Stored procedure group. Its purpose is to list procedures and validity. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SHOW PROCEDURES [ON table];
```

Example

```
SHOW PROCEDURES ON customers;
```

Operational notes

- Reports hash/tamper status.
- Use before CALL when diagnosing.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 48. DROP PROCEDURE

Field	Value
Category	Stored procedure
Purpose	Remove a stored procedure.
Related	CREATE PROCEDURE

DROP PROCEDURE belongs to the Stored procedure group. Its purpose is to remove a stored procedure. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
DROP PROCEDURE name ON table;
```

Example

```
DROP PROCEDURE list_customers ON customers;
```

Operational notes

- Requires FULL permission.
- Cannot be issued from inside a procedure.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 49. CALL

Field	Value
Category	Stored procedure
Purpose	Execute a stored procedure.
Related	CREATE PROCEDURE

CALL belongs to the Stored procedure group. Its purpose is to execute a stored procedure. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
CALL procedure_name(args...);
```

Example

```
CALL get_customer(1);
```

Operational notes

- Executes with invoker rights.
- Forbidden inside stored procedure bodies.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 50. DECLARE

Field	Value
Category	Procedure language
Purpose	Create a local variable.
Related	SET

DECLARE belongs to the Procedure language group. Its purpose is to create a local variable. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
DECLARE v_name TYPE DEFAULT value;
```

Example

```
DECLARE v_bucket STRING DEFAULT 'small';
```

Operational notes

- Procedure-body statement only.
- Variables are local to the call.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 51. SET

Field	Value
Category	Procedure language
Purpose	Assign a local variable.
Related	DECLARE

SET belongs to the Procedure language group. Its purpose is to assign a local variable. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SET v_name = expression;
```

Example

```
SET v_i = :v_i + 1;
```

Operational notes

- Input parameters are read-only.
- Simple numeric + and - support loop counters.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 52. IF

Field	Value
Category	Procedure language
Purpose	Start a conditional branch.
Related	ELSEIF

IF belongs to the Procedure language group. Its purpose is to start a conditional branch. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
IF condition THEN
```

Example

```
IF :p_amount >= 1000 THEN
```

Operational notes

- Use with ENDIF.
- Supports comparison and logic operators.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 53. ELSEIF

Field	Value
Category	Procedure language
Purpose	Add another conditional branch.
Related	ELSE

ELSEIF belongs to the Procedure language group. Its purpose is to add another conditional branch. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
ELSEIF condition THEN
```

Example

```
ELSEIF :p_amount >= 500 THEN
```

Operational notes

- Cannot appear after ELSE.
- Use for ordered thresholds.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 54. ELSE

Field	Value
Category	Procedure language
Purpose	Add fallback branch.
Related	ENDIF

ELSE belongs to the Procedure language group. Its purpose is to add fallback branch. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
ELSE
```

Example

```
ELSE
```

Operational notes

- Only one ELSE in an IF block.
- Runs when earlier branches are false.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 55. ENDIF

Field	Value
Category	Procedure language
Purpose	Close an IF block.
Related	IF

ENDIF belongs to the Procedure language group. Its purpose is to close an if block. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
ENDIF;
```

Example

```
ENDIF;
```

Operational notes

- Required for every IF.
- Control nesting is limited for safety.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 56. WHILE

Field	Value
Category	Procedure language
Purpose	Start a loop.
Related	ENDWHILE

WHILE belongs to the Procedure language group. Its purpose is to start a loop. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
WHILE condition DO
```

Example

```
WHILE :v_i <= :p_limit DO
```

Operational notes

- Move toward exit with SET.
- Loop iterations are bounded.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TigerDB Command 57. ENDWHILE

Field	Value
Category	Procedure language
Purpose	Close a WHILE block.
Related	WHILE

ENDWHILE belongs to the Procedure language group. Its purpose is to close a while block. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
ENDWHILE;
```

Example

```
ENDWHILE;
```

Operational notes

- Required for every WHILE.
- Nested WHILE depth is limited.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Keyword and Method Atlas

The TLang atlas summarizes the frontend language features used in TigerDB applications. The examples assume the Phase 4.4 convenience upgrade.

TLang Entry 1. SET SCREEN

Field	Value
Category	Screen
Purpose	Set terminal screen size.
Related	CLEAR SCREEN

SET SCREEN belongs to the Screen group. Its purpose is to set terminal screen size. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
SET SCREEN rows,cols
```

Example

```
SET SCREEN 24,80
```

Operational notes

- Use at program start.
- Keep forms within screen bounds.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 2. CLEAR SCREEN

Field	Value
Category	Screen
Purpose	Clear the display.
Related	@ SAY

CLEAR SCREEN belongs to the Screen group. Its purpose is to clear the display. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
CLEAR SCREEN
```

Example

```
CLEAR SCREEN
```

Operational notes

- Useful before drawing a form.
- Avoid clearing after every small update.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 3. @ SAY

Field	Value
Category	Screen
Purpose	Display a value at a position.
Related	@ GET

@ SAY belongs to the Screen group. Its purpose is to display a value at a position. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
@row,col SAY expression
```

Example

```
@3,5 SAY "Customer"
```

Operational notes

- Coordinates are row and column.
- Use tables for repeated layouts.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 4. @ GET

Field	Value
Category	Input
Purpose	Read a value into a variable.
Related	READ

@ GET belongs to the Input group. Its purpose is to read a value into a variable. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
@row,col GET variable
```

Example

```
@5,18 GET customer_id
```

Operational notes

- Usually followed by READ.
- Use labels near fields.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 5. READ

Field	Value
Category	Input
Purpose	Process pending input fields.
Related	@ GET

READ belongs to the Input group. Its purpose is to process pending input fields. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
READ
```

Example

```
READ
```

Operational notes

- Completes GET interaction.
- Validate inputs after READ.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 6. var

Field	Value
Category	Language
Purpose	Declare or assign a flexible value.
Related	FUNCTION

var belongs to the Language group. Its purpose is to declare or assign a flexible value. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
var name = expression
```

Example

```
var amount = 750.25
```

Operational notes

- Use meaningful names.
- Can store strings, numbers, booleans, JSON, or objects.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 7. FUNCTION

Field	Value
Category	Language
Purpose	Define reusable logic.
Related	RETURN

FUNCTION belongs to the Language group. Its purpose is to define reusable logic. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
FUNCTION name() ... ENDFUNCTION
```

Example

```
FUNCTION show_title()
  @1,1 SAY "Title"
ENDFUNCTION
```

Operational notes

- Keep functions small.
- Use for display and repeated workflows.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 8. RETURN

Field	Value
Category	Language
Purpose	Return from a function.
Related	FUNCTION

RETURN belongs to the Language group. Its purpose is to return from a function. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
RETURN expression
```

Example

```
RETURN amount
```

Operational notes

- Useful in calculations.
- Avoid returning connection objects unless intended.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 9. IF

Field	Value
Category	Control flow
Purpose	Start a conditional block.
Related	ELSEIF

IF belongs to the Control flow group. Its purpose is to start a conditional block. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
IF condition THEN
```

Example

```
IF amount > 100 THEN
```

Operational notes

- Use ENDIF.
- Keep conditions readable.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 10. ELSEIF

Field	Value
Category	Control flow
Purpose	Add conditional branch.
Related	ELSE

ELSEIF belongs to the Control flow group. Its purpose is to add conditional branch. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
ELSEIF condition THEN
```

Example

```
ELSEIF amount > 50 THEN
```

Operational notes

- Use for ordered alternatives.
- Place before ELSE.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 11. ELSE

Field	Value
Category	Control flow
Purpose	Fallback branch.
Related	ENDIF

ELSE belongs to the Control flow group. Its purpose is to fallback branch. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
ELSE
```

Example

```
ELSE
```

Operational notes

- Runs when earlier branches fail.
- Only one ELSE per IF block.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 12. ENDIF

Field	Value
Category	Control flow
Purpose	Close IF block.
Related	IF

ENDIF belongs to the Control flow group. Its purpose is to close if block. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
ENDIF
```

Example

```
ENDIF
```

Operational notes

- Required for IF.
- Improves readability when aligned.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 13. WHILE

Field	Value
Category	Control flow
Purpose	Repeat while condition is true.
Related	ENDWHILE

WHILE belongs to the Control flow group. Its purpose is to repeat while condition is true. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
WHILE condition DO
```

Example

```
WHILE i <= 10 DO
```

Operational notes

- Ensure loop changes state.
- Prefer FOREACH for rows.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 14. ENDWHILE

Field	Value
Category	Control flow
Purpose	Close WHILE loop.
Related	WHILE

ENDWHILE belongs to the Control flow group. Its purpose is to close while loop. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
ENDWHILE
```

Example

```
ENDWHILE
```

Operational notes

- Required for WHILE.
- Keep loop bodies short.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 15. FOREACH

Field	Value
Category	RecordSet
Purpose	Iterate rows in a RecordSet.
Related	ENDFOREACH

FOREACH belongs to the RecordSet group. Its purpose is to iterate rows in a recordset. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
FOREACH row IN rs
```

Example

```
FOREACH row IN customers
```

Operational notes

- Preferred for query results.
- Access fields with row["name"].
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 16. ENDFOREACH

Field	Value
Category	RecordSet
Purpose	Close FOREACH loop.
Related	FOREACH

ENDFOREACH belongs to the RecordSet group. Its purpose is to close foreach loop. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
ENDFOREACH
```

Example

```
ENDFOREACH
```

Operational notes

- Required for FOREACH.
- Increment display row variables inside loop.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 17. DEFINE TABLE

Field	Value
Category	UI
Purpose	Draw a table border.
Related	@ SAY

DEFINE TABLE belongs to the UI group. Its purpose is to draw a table border. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
DEFINE TABLE FROM r1,c1 TO r2,c2 DOUBLE
```

Example

```
DEFINE TABLE FROM 2,1 TO 12,78 DOUBLE
```

Operational notes

- Use for reports.
- Avoid overcrowding small screens.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 18. FORM

Field	Value
Category	UI
Purpose	Define a form.
Related	GET

FORM belongs to the UI group. Its purpose is to define a form. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
DEFINE FORM ... ENDFORM
```

Example

```
DEFINE FORM customer_form ... ENDFORM
```

Operational notes

- Group related fields.
- Keep validation separate.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 19. MENU

Field	Value
Category	UI
Purpose	Define menu actions.
Related	POPUP

MENU belongs to the UI group. Its purpose is to define menu actions. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
DEFINE MENU name
```

Example

```
DEFINE MENU main
```

Operational notes

- Good for application navigation.
- Use clear prompts.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 20. POPUP

Field	Value
Category	UI
Purpose	Define a choice popup.
Related	MENU

POPUP belongs to the UI group. Its purpose is to define a choice popup. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
DEFINE POPUP name
```

Example

```
DEFINE POPUP choices
```

Operational notes

- Useful for small lists.
- Avoid too many choices.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 21. CONNECT TIGERDB SERVER

Field	Value
Category	Database
Purpose	Connect directly to TigerDB Server.
Related	CONNECT TIGERDB ROUTER

CONNECT TIGERDB SERVER belongs to the Database group. Its purpose is to connect directly to tigerdb server. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
CONNECT TO TIGERDB SERVER HOST "h" ON PORT 9191 USER "u" PASSWORD ENV "P" AS PD
```

Example

```
CONNECT TO TIGERDB SERVER HOST "127.0.0.1" ON PORT 9191 USER "admin" PASSWORD ENV "TIGERDB_PASSWORD" AS PD
```

Operational notes

- Direct server mode.
- Backend port is 9191 by default.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 22. CONNECT TIGERDB ROUTER

Field	Value
Category	Database
Purpose	Connect through TigerDB Router.
Related	PD.query

CONNECT TIGERDB ROUTER belongs to the Database group. Its purpose is to connect through tigerdb router. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
CONNECT TO TIGERDB ROUTER HOST "h" ON PORT 9292 USER "u" PASSWORD ENV "P" AS PD
```

Example

```
CONNECT TO TIGERDB ROUTER HOST "127.0.0.1" ON PORT 9292 USER "admin" PASSWORD ENV "TIGERDB_PASSWORD" DATABASE "sales" AS PD
```

Operational notes

- Router port is 9292 by default.
- Database name maps to backend route.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 23. PASSWORD ENV

Field	Value
Category	Security
Purpose	Read password from environment variable.
Related	ENV

PASSWORD ENV belongs to the Security group. Its purpose is to read password from environment variable. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
PASSWORD ENV "VARIABLE"
```

Example

```
PASSWORD ENV "TIGERDB_PASSWORD"
```

Operational notes

- Avoid hard-coded passwords.
- Set environment before running executable.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 24. ENV

Field	Value
Category	Function
Purpose	Read an environment variable.
Related	PASSWORD ENV

ENV belongs to the Function group. Its purpose is to read an environment variable. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
ENV("NAME")
```

Example

```
var host = ENV("TIGERDB_HOST")
```

Operational notes

- Returns environment value.
- Use defaults when empty.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 25. PD.query

Field	Value
Category	Database method
Purpose	Run SQL and return raw JSON.
Related	PD.queryRS

PD.query belongs to the Database method group. Its purpose is to run sql and return raw json. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
PD.query(sql)
```

Example

```
var raw = PD.query("SELECT * FROM customers")
```

Operational notes

- Use OK and ERROR for raw responses.
- Use queryRS when rows are needed.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 26. PD.queryRS

Field	Value
Category	Database method
Purpose	Run SQL and return RecordSet.
Related	FOREACH

PD.queryRS belongs to the Database method group. Its purpose is to run sql and return recordset. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
PD.queryRS(sql)
```

Example

```
RecordSet rs = PD.queryRS("SELECT * FROM customers")
```

Operational notes

- Convenience wrapper around query + TO_RS.
- Use with FOREACH.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 27. PD.execute

Field	Value
Category	Database method
Purpose	Run SQL where rows are not needed.
Related	PD.query

PD.execute belongs to the Database method group. Its purpose is to run sql where rows are not needed. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
PD.execute(sql)
```

Example

```
PD.execute("CREATE TABLE t (id INT PRIMARY KEY)")
```

Operational notes

- Good for DDL and writes.
- Check response in real applications.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 28. PD.call

Field	Value
Category	Database method
Purpose	Call stored procedure and return raw JSON.
Related	PD.callRS

PD.call belongs to the Database method group. Its purpose is to call stored procedure and return raw json. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
PD.call(name,args...)
```

Example

```
var r = PD.call("record_sale", 1, 101, 750.25)
```

Operational notes

- Use OK/ERROR.
- Arguments are converted to SQL literals.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 29. PD.callRS

Field	Value
Category	Database method
Purpose	Call stored procedure and return RecordSet.
Related	PD.call

PD.callRS belongs to the Database method group. Its purpose is to call stored procedure and return recordset. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
PD.callRS(name,args...)
```

Example

```
RecordSet rs = PD.callRS("monthly_sales", "2026-04")
```

Operational notes

- Use when procedure returns rows.
- Iterate with FOREACH.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 30. TO_RS

Field	Value
Category	RecordSet
Purpose	Convert raw JSON result to RecordSet.
Related	PD.queryRS

TO_RS belongs to the RecordSet group. Its purpose is to convert raw json result to recordset. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
TO_RS(json)
```

Example

```
RecordSet rs = TO_RS(raw)
```

Operational notes

- Useful with PD.query.
- queryRS is shorter.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 31. OK

Field	Value
Category	JSON helper
Purpose	Check whether JSON response is successful.
Related	ERROR

OK belongs to the JSON helper group. Its purpose is to check whether json response is successful. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
OK(json)
```

Example

```
IF OK(result) THEN
```

Operational notes

- Looks for ok:false.
- Use with raw JSON.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 32. ERROR

Field	Value
Category	JSON helper
Purpose	Extract error text from JSON response.
Related	OK

ERROR belongs to the JSON helper group. Its purpose is to extract error text from json response. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
ERROR(json)
```

Example

```
@ 10,1 SAY ERROR(result)
```

Operational notes

- Use in failure displays.
- May fall back to message.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 33. row["field"]

Field	Value
Category	RecordSet
Purpose	Access a field from a row.
Related	FOREACH

row["field"] belongs to the RecordSet group. Its purpose is to access a field from a row. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
row["column_name"]
```

Example

```
@rownum,1 SAY row["name"]
```

Operational notes

- Field names match JSON keys.
- Use inside FOREACH.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

TLang Entry 34. PD.close

Field	Value
Category	Database method
Purpose	Close the connection.
Related	CONNECT

PD.close belongs to the Database method group. Its purpose is to close the connection. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
PD.close()
```

Example

```
PD.close()
```

Operational notes

- Run before exit.
- Important for Router sessions.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

Router Command and Configuration Atlas

The Router atlas summarizes command and configuration entries from the current R8 operational line. The Router remains a gateway and does not execute stored-procedure bodies.

Router Entry 1. ROUTER STATUS

Field	Value
Category	Router admin
Purpose	Show Router statistics and state.
Related	ROUTER BACKENDS

ROUTER STATUS belongs to the Router admin group. Its purpose is to show router statistics and state. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
ROUTER STATUS;
```

Example

```
ROUTER STATUS;
```

Operational notes

- Admin permission required.
- Use after restart and after reload.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

Router Entry 2. ROUTER BACKENDS

Field	Value
Category	Router admin
Purpose	Show backend status and health metrics.
Related	ROUTER HEALTH

ROUTER BACKENDS belongs to the Router admin group. Its purpose is to show backend status and health metrics. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
ROUTER BACKENDS;
```

Example

```
ROUTER BACKENDS;
```

Operational notes

- Includes health and latency fields.
- Check after backend route changes.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

Router Entry 3. ROUTER DATABASES

Field	Value
Category	Router admin
Purpose	Show database routes.
Related	DATABASE config lines

ROUTER DATABASES belongs to the Router admin group. Its purpose is to show database routes. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
ROUTER DATABASES;
```

Example

```
ROUTER DATABASES;
```

Operational notes

- Shows route status.
- Useful after ROUTER RELOAD.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

Router Entry 4. ROUTER POOLS

Field	Value
Category	Router admin
Purpose	Show backend pool state.
Related	BACKEND_POOL

ROUTER POOLS belongs to the Router admin group. Its purpose is to show backend pool state. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
ROUTER POOLS;
```

Example

```
ROUTER POOLS;
```

Operational notes

- Helps diagnose idle sessions.
- Pool settings are read safely in R8.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

Router Entry 5. ROUTER HEALTH

Field	Value
Category	Router admin
Purpose	Force live backend health checks.
Related	ROUTER HEALTH STATUS

ROUTER HEALTH belongs to the Router admin group. Its purpose is to force live backend health checks. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
ROUTER HEALTH;
```

Example

```
ROUTER HEALTH;
```

Operational notes

- Performs network checks.
- May take longer than cached status.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

Router Entry 6. ROUTER HEALTH STATUS

Field	Value
Category	Router admin
Purpose	Show cached backend health status.
Related	ROUTER HEALTH

ROUTER HEALTH STATUS belongs to the Router admin group. Its purpose is to show cached backend health status. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
ROUTER HEALTH STATUS;
```

Example

```
ROUTER HEALTH STATUS;
```

Operational notes

- No forced network ping.
- Good for dashboards.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

Router Entry 7. ROUTER CLIENTS

Field	Value
Category	Router admin
Purpose	Show active client/session metadata.
Related	ROUTER SESSIONS

ROUTER CLIENTS belongs to the Router admin group. Its purpose is to show active client/session metadata. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
ROUTER CLIENTS;
```

Example

```
ROUTER CLIENTS;
```

Operational notes

- Does not expose passwords.
- Alias: ROUTER SESSIONS.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

Router Entry 8. ROUTER SESSIONS

Field	Value
Category	Router admin
Purpose	Alias for ROUTER CLIENTS.
Related	ROUTER CLIENTS

ROUTER SESSIONS belongs to the Router admin group. Its purpose is to alias for router clients. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
ROUTER SESSIONS;
```

Example

```
ROUTER SESSIONS;
```

Operational notes

- Use whichever term your team prefers.
- Same output as clients.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

Router Entry 9. ROUTER RELOAD

Field	Value
Category	Router admin
Purpose	Reload config without restart.
Related	SIGHUP

ROUTER RELOAD belongs to the Router admin group. Its purpose is to reload config without restart. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
ROUTER RELOAD;
```

Example

```
ROUTER RELOAD;
```

Operational notes

- LISTEN_HOST and LISTEN_PORT still require restart.
- Invalid config leaves current config active.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

Router Entry 10. router_status JSON

Field	Value
Category	JSON action
Purpose	Request status through JSON.
Related	ROUTER STATUS

router_status JSON belongs to the JSON action group. Its purpose is to request status through json. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
{ "action": "router_status" }
```

Example

```
{ "action": "router_status" }
```

Operational notes

- Admin permission required.
- Useful for programmatic dashboards.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

Router Entry 11. router_clients JSON

Field	Value
Category	JSON action
Purpose	Request active sessions through JSON.
Related	ROUTER CLIENTS

router_clients JSON belongs to the JSON action group. Its purpose is to request active sessions through json. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
{ "action": "router_clients" }
```

Example

```
{ "action": "router_clients" }
```

Operational notes

- Admin permission required.
- Alias: router_sessions.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

Router Entry 12.router_health_check JSON

Field	Value
Category	JSON action
Purpose	Force backend health checks through JSON.
Related	ROUTER HEALTH

router_health_check JSON belongs to the JSON action group. Its purpose is to force backend health checks through json. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
{ "action": "router_health_check" }
```

Example

```
{ "action": "router_health_check" }
```

Operational notes

- Admin permission required.
- Returns detailed backend status.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

Router Entry 13.AUTH_THROTTLE

Field	Value
Category	Router config
Purpose	Enable login throttling.
Related	AUTH_FAIL_THRESHOLD

AUTH_THROTTLE belongs to the Router config group. Its purpose is to enable login throttling. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
AUTH_THROTTLE=ON
```

Example

```
AUTH_THROTTLE=ON
AUTH_FAIL_THRESHOLD=5
```

Operational notes

- Protects against repeated bad logins.
- Returns retry_after_seconds instead of sleeping.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

Router Entry 14. BACKEND_POOL

Field	Value
Category	Router config
Purpose	Enable backend connection pooling.
Related	ROUTER POOLS

BACKEND_POOL belongs to the Router config group. Its purpose is to enable backend connection pooling. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
BACKEND_POOL=ON
```

Example

```
BACKEND_POOL=ON
BACKEND_POOL_MAX_PER_BACKEND=16
```

Operational notes

- Improves reuse.
- Flushes stale pools after reload.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

Router Entry 15.

STORED_PROCEDURE_CALL_ROLE

Field	Value
Category	Router config
Purpose	Classify CALL forwarding policy.
Related	CALL

STORED_PROCEDURE_CALL_ROLE belongs to the Router config group. Its purpose is to classify call forwarding policy. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
STORED_PROCEDURE_CALL_ROLE=FULL
```

Example

```
STORED_PROCEDURE_CALL_ROLE=READ
STORED_PROCEDURE_CALL_REQUIRE_SAME_BACKEND_USER=ON
```

Operational notes

- Default FULL is conservative.
- READ should require same backend user.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

Router Entry 16. BACKEND line

Field	Value
Category	Router config
Purpose	Define a backend TigerDB server.
Related	DATABASE line

BACKEND line belongs to the Router config group. Its purpose is to define a backend tigerdb server. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
BACKEND name host port
```

Example

```
BACKEND main 127.0.0.1 9191
```

Operational notes

- Names are used by DATABASE routes.
- Keep backend ports private.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

Router Entry 17. DATABASE line

Field	Value
Category	Router config
Purpose	Route a database to a backend.
Related	BACKEND line

DATABASE line belongs to the Router config group. Its purpose is to route a database to a backend. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
DATABASE db backend
```

Example

```
DATABASE sales main
```

Operational notes

- Applications use the database name.
- Reload after edits.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

Router Entry 18. USER line

Field	Value
Category	Router config
Purpose	Define Router and backend credentials.
Related	GRANT line

USER line belongs to the Router config group. Its purpose is to define router and backend credentials. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
USER router_user router_pass backend_user backend_pass
```

Example

```
USER app apppass app apppass
```

Operational notes

- Protect config with OS permissions.
- Avoid mapping normal users to backend admin.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.

Router Entry 19. GRANT line

Field	Value
Category	Router config
Purpose	Assign Router database role.
Related	USER line

GRANT line belongs to the Router config group. Its purpose is to assign router database role. In a production workflow, run it first in a small database, inspect the JSON response, and then place it into a script, TLang program, driver method, or Router-managed flow.

The safest way to learn this entry is to separate intent from syntax. First decide what object the command affects, then check what permission is needed, and finally verify the result with a SHOW or SELECT command.

Syntax

```
GRANT username database ROLE
```

Example

```
GRANT report sales READ
```

Operational notes

- Roles are READ, WRITE, FULL.
- Use least privilege.
- Record the exact command form used by the application team.
- Prefer a tested smoke command over a large first production run.



About the author

With a career spanning over 25 years, the author is a seasoned expert at the intersection of technology and finance. His academic foundation is built on a rigorous trifecta of disciplines: a degree in Mathematics, a Master's in Computer Science (MCS), and an MBA in Finance.

He is perhaps best known as the developer of the TigerDB database server, a testament to his deep expertise in complex systems architecture. Throughout his professional journey, he has served as a strategic consultant and corporate trainer for some of the world's most prestigious multinational organizations. His portfolio includes high-impact work with IT giants like IBM and top-tier financial institutions such as J.P. Morgan Chase, BNP Paribas, UBS, Franklin Templeton, and Nomura.

Blending technical mastery with executive insight, he brings a wealth of hands-on experience in software development and high-level consultancy to his writing, offering readers a unique perspective forged in the fast-paced world of global enterprise.