Centre de Calcul de l'Institut National de Physique Nucléaire et de Physique des Particules

SQL Databases Training

January 22 -25, 2019

Osman Aidel – Philippe Cheynet

CCIN2P3

cnrs

IN2P3

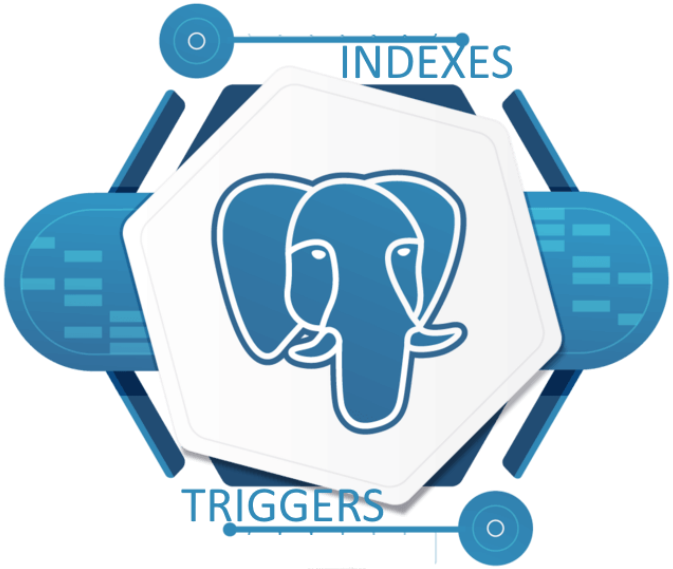
Course Content

- DB history (30 min)
- Relational Model (3h)
- SQL (3h)
- PL/SQL (4h)
- Advanced Objects (3h)
- Optimization (7h)

Course Content

- DB history (30 min)
- Relational Model (3h)
- SQL (3h)
- Advanced Objects (3h)
- PL/SQL (4h)
- Optimization (7h)

- WHY / HOW TO (SQL Architecture)
- Views
  - Complex Views
  - Materialized Views
- Indexes
  - B-tree, hash, GiST, SP-GiST, GIN, and BRIN
  - Index Organized Data (cluster)
- Triggers
- Prepared Statements
- Partitionning



VIEW

A view:

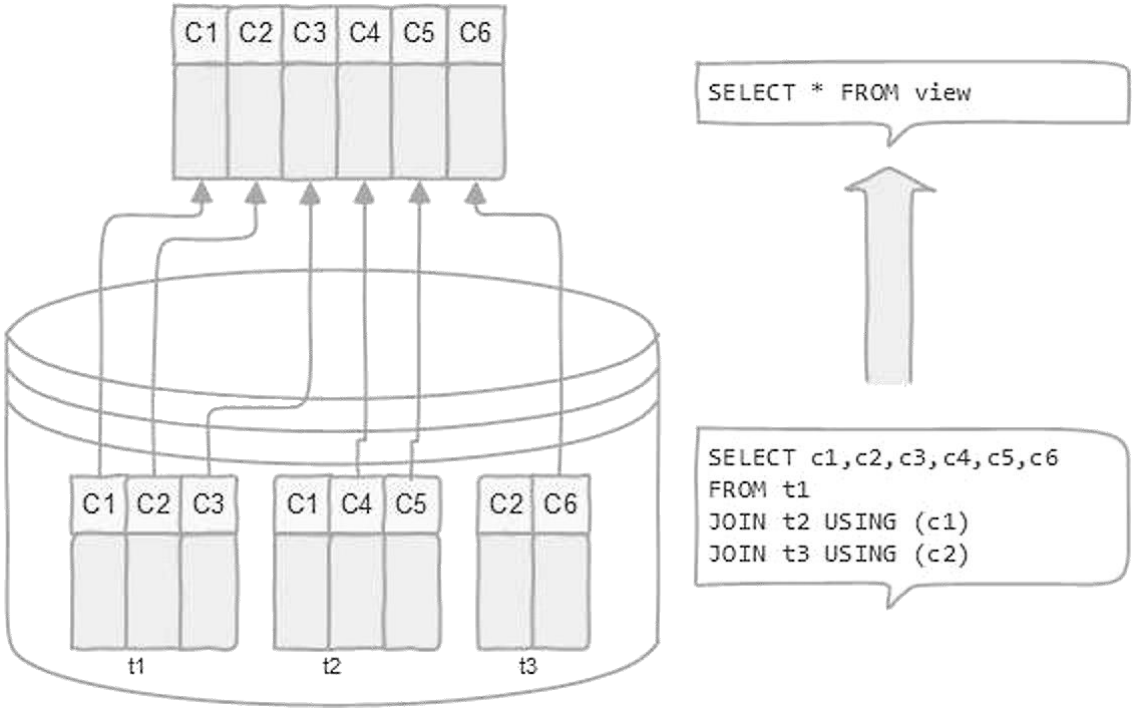
- is a named stored query that provides another way to present data in the database tables
- can be accessed as a virtual table
- does not store data physically / no more space [except for one type]
- allows the user to more easily find relevant information in large datasets.
- allows to show certain data points while hiding others.
- returns a virtual table containing just the information you want to see

A view is defined based on one or more tables, which are known as base tables

VIEW :

A view can be very useful in some cases such as:

- A view helps simplify the complexity of a query because you can query a view, which is based on a complex query, using a simple SELECT statement.
- Like a table, you can grant permission to users through a view that contains specific data that the users are authorized to see.
- A view provides a consistent layer even if the columns of underlying table changes.



There are 2 types of Views in SQL:

SIMPLE VIEW	COMPLEX VIEW
Contains only one single base table or is created from only one table.	Contains more than one base tables or is created from more than one tables.
Disallows the use of group functions like MAX(), COUNT(), etc.	Allows the use of use group functions.
Does not contain groups of data.	Can contain groups of data.
DML operations could be performed through a simple view.	DML operations could not always be performed through a complex view.
INSERT, DELETE and UPDATE are directly possible on a simple view.	INSERT, DELETE and UPDATE can be applied on complex view directly.
Does not contain group by, distinct, pseudocolumn like rownum, columns defiend by expressions.	It can contain group by, distinct, pseudocolumn like rownum, columns defiend by expressions.
Does not include NOT NULL columns from base tables.	NOT NULL columns can be included in complex view.

Updating a view

Simple views are automatically updatable: the system will allow INSERT, UPDATE and DELETE statements to be used on the view in the same way as on a regular table.

If the view is automatically updatable the system will convert any data modification statement on the view into the corresponding statement on the underlying base relation.

An updatable view may contain a mix of updatable and non-updatable columns. A column is updatable if it is a simple reference to an updatable column of the underlying base relation.

A more complex view that does not satisfy all these conditions is read-only by default: the system will not allow an insert, update, or delete on the view. You can get the effect of an updatable view by creating INSTEAD OF TRIGGERS on the view, which convert attempted inserts, etc. on the view into appropriate actions on other tables.

With Check Option views

If an automatically updatable view contains a WHERE condition, the condition restricts which rows of the base relation are available to be modified by UPDATE and DELETE statements on the view.
However, an UPDATE is allowed to change a row so that it no longer satisfies the WHERE condition, and thus is no longer visible through the view.
Similarly, an INSERT command can potentially insert base-relation rows that do not satisfy the WHERE condition and thus are not visible through the view.

We may grant the permission to the users to update a limited set of rows of the base table

WITH CHECK OPTION

SYNTAX

```
CREATE [ OR REPLACE ] [TEMPORARY] VIEW name [ ( column_name [, ...] ) ]
AS query
[ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

- **CASCADED** — New rows are checked against the conditions of the view and all underlying base views. Default Value
- **LOCAL** — New rows are only checked against the conditions defined directly in the view itself. Any conditions defined on underlying base views are not checked

```
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] … …
ALTER VIEW [ IF EXISTS ] name RENAME TO new_name
```

```
DROP VIEW [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

MATERIALIZED VIEWS

Special views called materialized views store data physically and refresh the data periodically from the base tables.
The materialized views have many advantages:

- faster access to data from a remote server,
- cache the result of a complex expensive query in data warehouses or business intelligence applications.

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] name
AS query
WITH [NO] DATA;
```

```
DROP MATERIALIZED VIEW [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

```
REFRESH MATERIALIZED VIEW [CONCURRENTLY] name;
```

Temporary updated version of the materialized view is created,
PostgreSQL compares two versions, and performs INSERT and UPDATE only on the differences.
You can query against the materialized view while it is being updated.

PosgreSQL locks the entire table
You cannot query data against it.

SQL – Advanced Objects - Indexes

Indexes are a common way to enhance database performance.

An index allows the database server to find and retrieve specific rows much faster than it could do without an index.

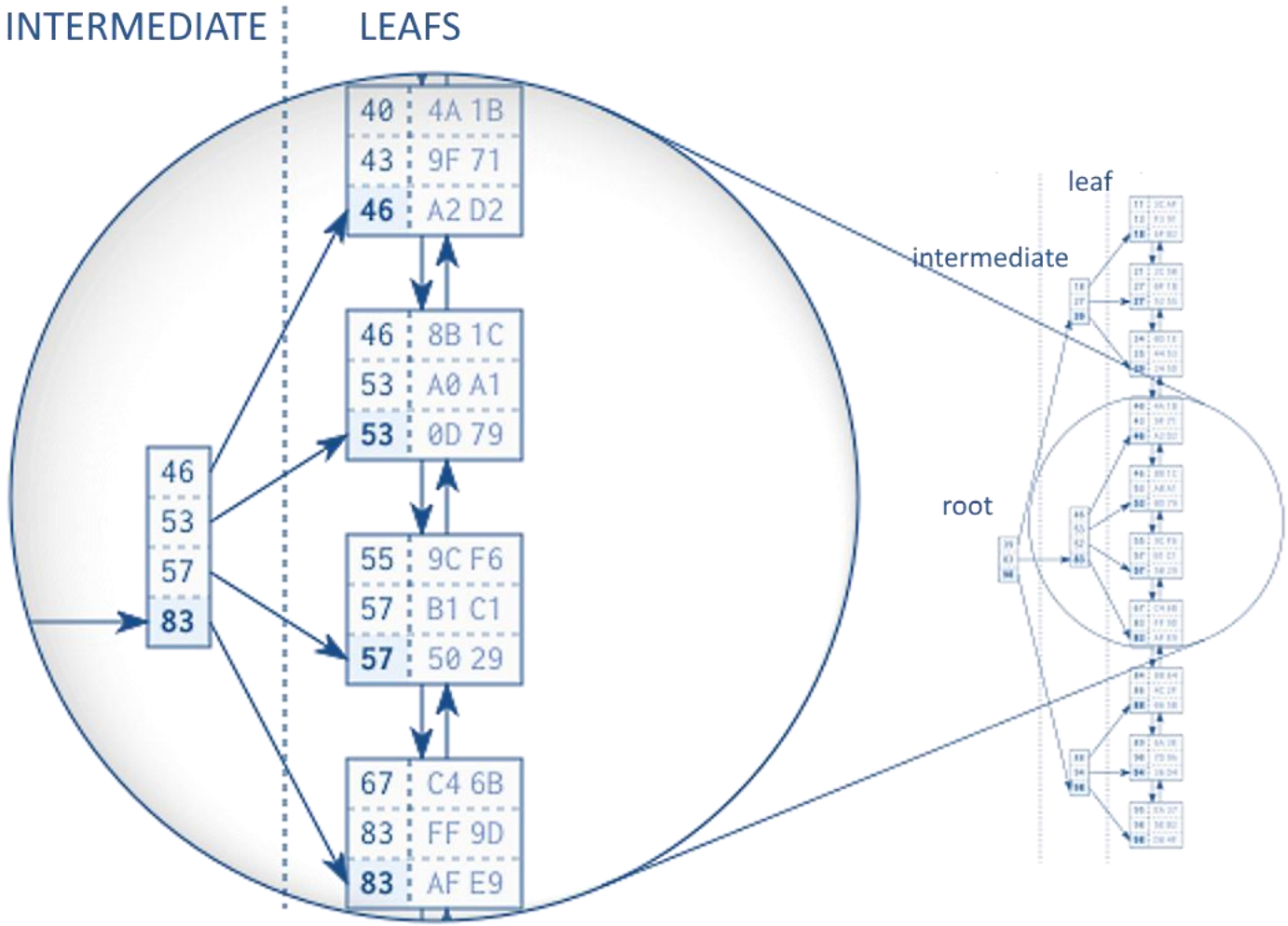
However, indexes add write and storage overheads to the database system, therefore, using them appropriately is very important

An index is a separated data structure that speeds up the data retrieval on a table at the cost of additional writes and storage to maintain it.

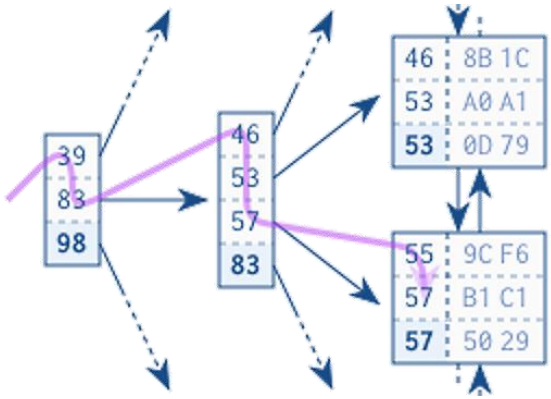
PostgreSQL provides several index types:
B-tree, Hash, GiST, SP-GiST, GIN and BRIN.

Each index type uses a different algorithm that is best suited to different types of queries.

B-tree Indexes



B-tree Indexes



Child nodes to the left of value “X” have values smaller than X; child nodes to the right of the value “X” have values greater than X

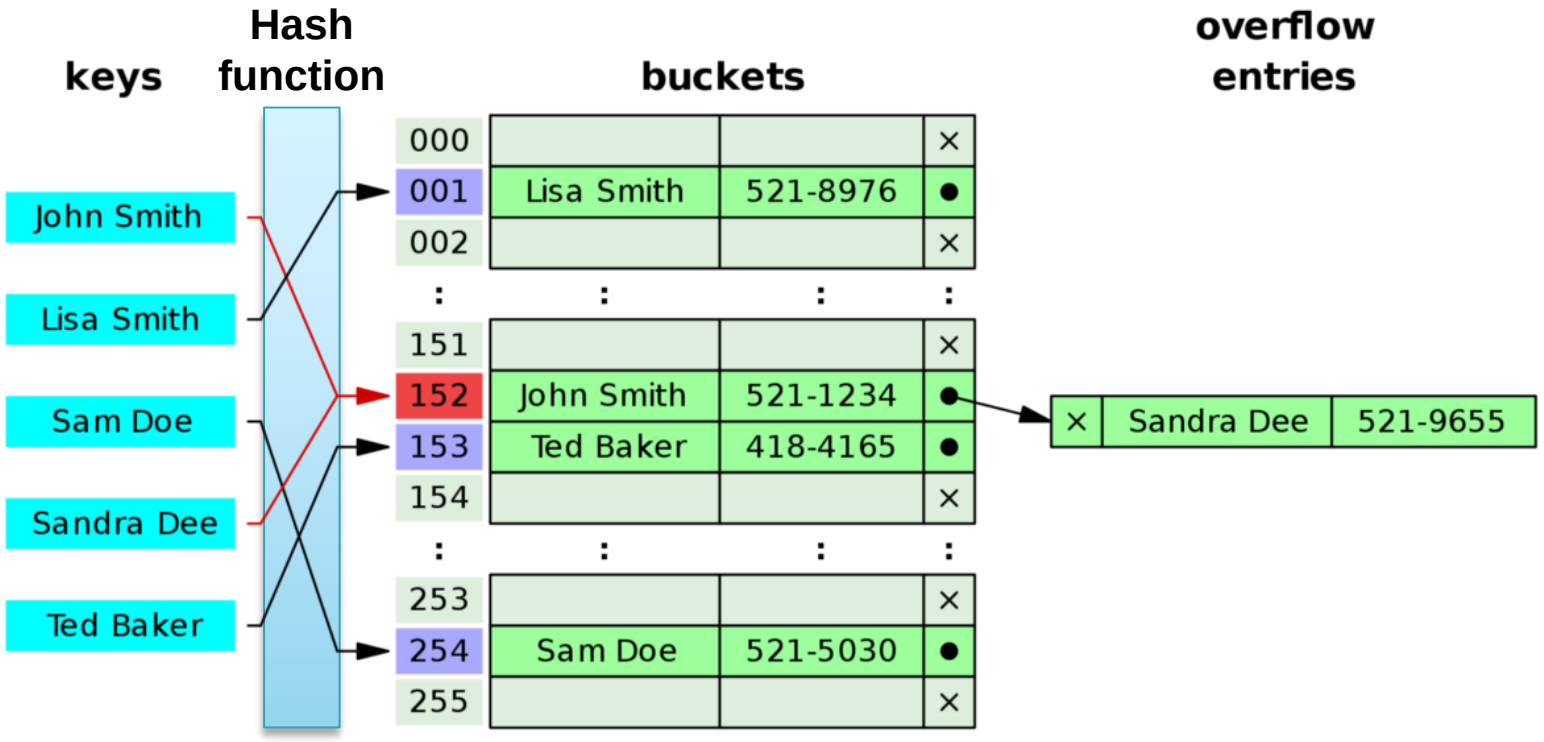
B-tree Indexes

Adding and removing values from a B-tree usually does not create new nodes: the number of values in each node can vary. Of course, that means we'll have some empty space, so a B-tree will require more disk space than a denser tree would.

Adding a value has to maintain both the order of value and the balance of the tree. First we'll find the leaf node where the values should be added. If there is enough space in the leaf node, we'll simply add the value; the structure and the tree depth won't change. We would split the parent node in the same manner if an overflow happens there. In extreme cases, we'd have to split the root node and the tree depth would increase.

To delete a value from the B-Tree, we'll locate that value and remove it. If that deletion causes underflow (the number of values stored in a node is too low) we'll have to merge nodes together.

Hash Indexes



When we want to **search** for a value, we'll use the hash function to calculate the address where our data could be stored. We'll look for the data in the bucket. If we find it, we're done. If we don't find our value, it means it's not in the index.

Hash Indexes

Adding new values works similarly: we'll use the hash function to calculate the address where we'll store our data. If that address is already occupied, we'll add new buckets and re-compute the hash function. Once again, we'll use the whole key as an input for our function. The result is the actual address (in disk memory) where we can find the desired data.

Updating or deleting values consists of first searching for a value and then applying the desired operation on that memory address.

Hash table indexes are very fast when testing for equality (= or <>). This is because we're using the whole key and not just its parts.
Individual parts can't help us when we want to find range (< or >)

Other Index Types

GiST indexes are not a single kind of index, but rather an infrastructure within which many different indexing strategies can be implemented. GiST indexes are also capable of optimizing “nearest-neighbor” searches, such as

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

which finds the ten places closest to a given target point.
-> geometric data types and full-text search

SP-GiST indexes, like GiST indexes, offer an infrastructure that supports various kinds of searches. SP-GiST permits implementation of a wide range of different non-balanced data structures, such as quadtrees, k-d trees, and radix trees
-> multimedia, phone routing, and IP routing

Other Index Types

GIN indexes are “inverted indexes” which are appropriate for data values that contain multiple component values, such as arrays. An inverted index contains a separate entry for each component value, and can efficiently handle queries that test for the presence of specific component values.
-> hstore, array, jsonb, and range types

BRIN indexes (a shorthand for Block Range INdexes) store summaries about the values stored in consecutive physical block ranges of a table.
For data types that have a linear sort order, the indexed data corresponds to the minimum and maximum values of the values in the column for each block range.
allows the use of an index on a very large table that would previously be impractical using B-tree
-> created date column of the sales order table.

A simple version of CREATE INDEX statement is as follows:

```
CREATE INDEX  index_name  ON table_name
[USING method]
(
    column_name [ASC | DESC] [NULL {FIRST | LAST }],
    ...
);
```

In this syntax:

- Index name after the CREATE INDEX clause.
  The index name should be meaningful and easy to remember.
- Name of the table to which the index belongs.
- Index method such as btree, hash, gist, spgist, gin, and brin.
  PostgreSQL uses btree by default.
- List one or more columns that has to be stored in the index.
  The ASC and DESC specify the sort order. ASC is the default.
  NULLS FIRST or NULLS LAST specifies nulls sort before or after non-nulls.
  NULLS FIRST is the default when DESC is specified
  NULLS LAST is the default when DESC is not specified.

DROP INDEX command syntax is:

```
DROP INDEX  [ CONCURRENTLY]
[ IF EXISTS ]  index_name
[ CASCADE | RESTRICT ];
```

CONCURRENTLY

When you execute the DROP INDEX statement, PostgreSQL acquires an exclusive lock on the table and blocks other accesses until the index removal completes. To force the command waits until the conflicting transaction completes before removing the index, you can use the CONCURRENTLY option.

CASCADE

If the index has dependent objects, use the CASCADE option to automatically drop these objects and all objects that depends on those objects.

RESTRICT

The RESTRICT option instructs PostgreSQL to refuse to drop the index if any objects depend on it. The DROP INDEX uses RESTRICT by default.

In practice, an index can become corrupted and no longer contains valid data. To recover the index, you can use the REINDEX statement:

```
REINDEX [ ( VERBOSE ) ] { INDEX | TABLE | SCHEMA | DATABASE | SYSTEM } name;
```

To recreate a single index, you specify the index name after REINDEX INDEX clause as follows:

```
REINDEX INDEX index_name;
REINDEX TABLE table_name;
REINDEX DATABASE database_name;
```

REINDEX
Locks writes but not reads of the table
Locks reads & write that attempt to use the index

VS

DROP INDEX
Exclusive lock on the table (locks both writes and reads of the table)
CREATE INDEX
Locks out writes but not reads

Cluster

CLUSTER instructs PostgreSQL to cluster a table based on an index.
The index must already have been defined on the table.

```
CLUSTER [VERBOSE] table_name [ USING index_name ]
```

When a table is clustered, it is physically reordered based on the index information.
Clustering is a one-time operation: when the table is subsequently updated, the changes are not clustered.

CLUSTER

Note : When a table is being clustered, an ACCESS EXCLUSIVE lock is acquired on it (both reads and writes).

INDEX-ORGANIZED TABLE

CLUSTERED INDEXES (INNODB)

SQL – Advanced Objects - Triggers

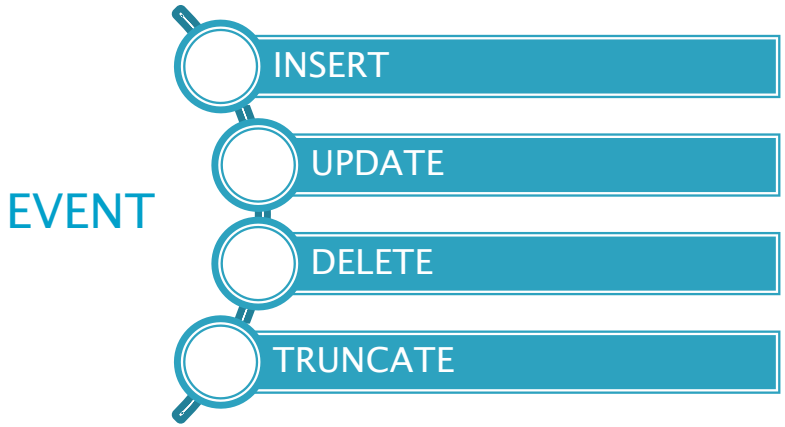
A trigger is a function invoked automatically whenever an event associated with a TABLE occurs.

For example, if you want to keep history of data without requiring application to have logic to check for every event.

You can also use triggers to maintain complex data integrity rules which you cannot implement elsewhere except at the database level.

To create a new trigger :
1. define a trigger function using CREATE FUNCTION statement.
2. bind this trigger function to a table using CREATE TRIGGER statement.

Trigger Characteristics:



UPDATE statement that affects 20 rows

ROW LEVEL
the trigger will be invoked 20 times

STATEMENT LEVEL
the trigger will be invoked 1 time

BEFORE TRIGGER
Can skip the row,
Modify values, …

→ EVENT →

AFTER TRIGGER
all changes are available

Trigger

On VIEWS, triggers can be defined to execute instead of INSERT, UPDATE, or DELETE operations. INSTEAD OF triggers may only be defined on views, and only at row level.

Such INSTEAD OF triggers are fired once for each row that needs to be modified in the view.

It is the responsibility of the trigger's function to perform the necessary modifications to the view's underlying base table(s) and, where appropriate, return the modified row as it will appear in the view.

Triggers on views can also be defined to execute once per SQL statement, before or after INSERT, UPDATE, or DELETE operations. However, such triggers are fired only if there is also an INSTEAD OF trigger on the view. Otherwise, any statement targeting the view must be rewritten into a statement affecting its underlying base table(s), and then the triggers that will be fired are the ones attached to the base table(s).

Trigger

Creating a trigger

```
CREATE TRIGGER trigger_name {BEFORE | AFTER | INSTEAD OF} {event [OR ...]}
   ON table_name
   [FOR [EACH] {ROW | STATEMENT}]
   EXECUTE PROCEDURE trigger_function
```

Modifying a trigger

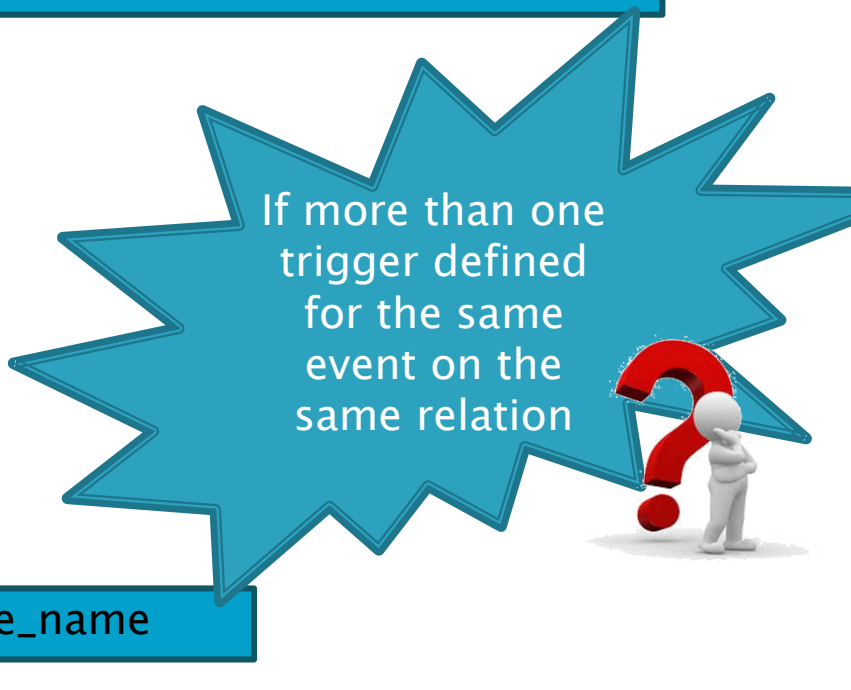
```
ALTER TRIGGER trigger_name ON table_name
RENAME TO new_name
```

Disabling a trigger

```
ALTER TABLE table_name
DISABLE TRIGGER trigger_name | ALL
```

Removing a trigger

```
DROP TRIGGER [IF EXISTS] trigger_name ON table_name
```

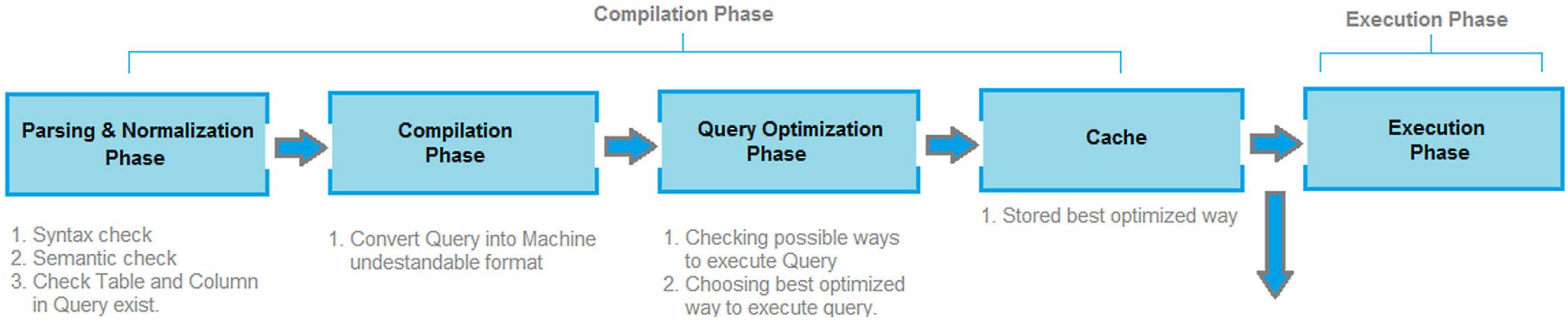


A PREPARED STATEMENT is a server-side object that can be used to optimize performance.

When creating a prepared statement, parameters are referred by position, using $1, $2, etc.

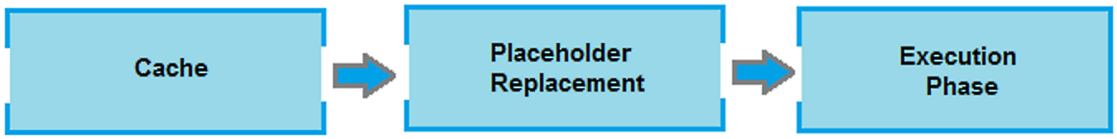
```
PREPARE seniorPerCountry (text, bigint) AS
    SELECT country, count(*),avg(age)
    FROM customers WHERE country = $1 and age > $2
    GROUP BY country;
```

When a statement is PREPARED, it is parsed, analyzed, and rewritten according to the compilation phase:



When an EXECUTE command is subsequently issued, the prepared statement is planned and executed.

```
EXECUTE seniorPerCountry('France',60);
EXECUTE seniorPerCountry('Australia',75);
```



This division of labor avoids repetitive parse analysis work, while allowing the execution plan to depend on the specific parameter values supplied.

SQL – Advanced Objects - Prepared Statements

Prepared statements only last for the duration of the current database session.

This also means that a single prepared statement cannot be used by multiple simultaneous database clients; however, each client can create its own prepared statement to use.

Prepared statements can be manually cleaned up using the DEALLOCATE command.

```
DEALLOCATE [ PREPARE ] { name | ALL }
```

SQL – Advanced Objects - Partitioning

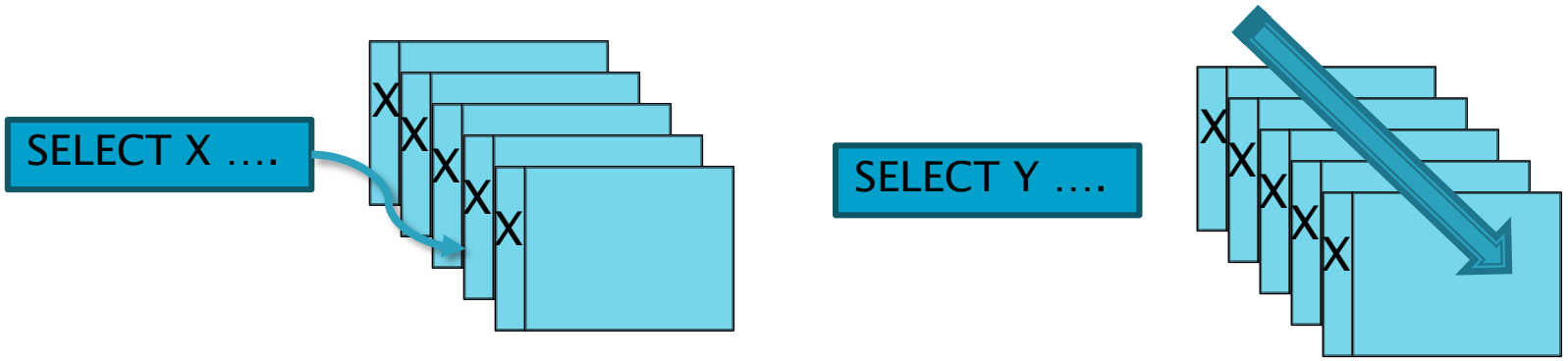
"As time goes by, the velocity and volume of the data increase more and more and queries become slower and slower as whole tables need to be scanned.“

But what happens in cases there is no need for a full scan?

Imagine the compilation of monthly business intelligence reports. The only data that are actually needed are those that were produced during the last month.

Before creating any partitions you should try to exhaust all other alternative options including table indexing and revision of queries.

The exact point at which a table will benefit from partitioning depends on the application



SQL – Advanced Objects - Partitioning

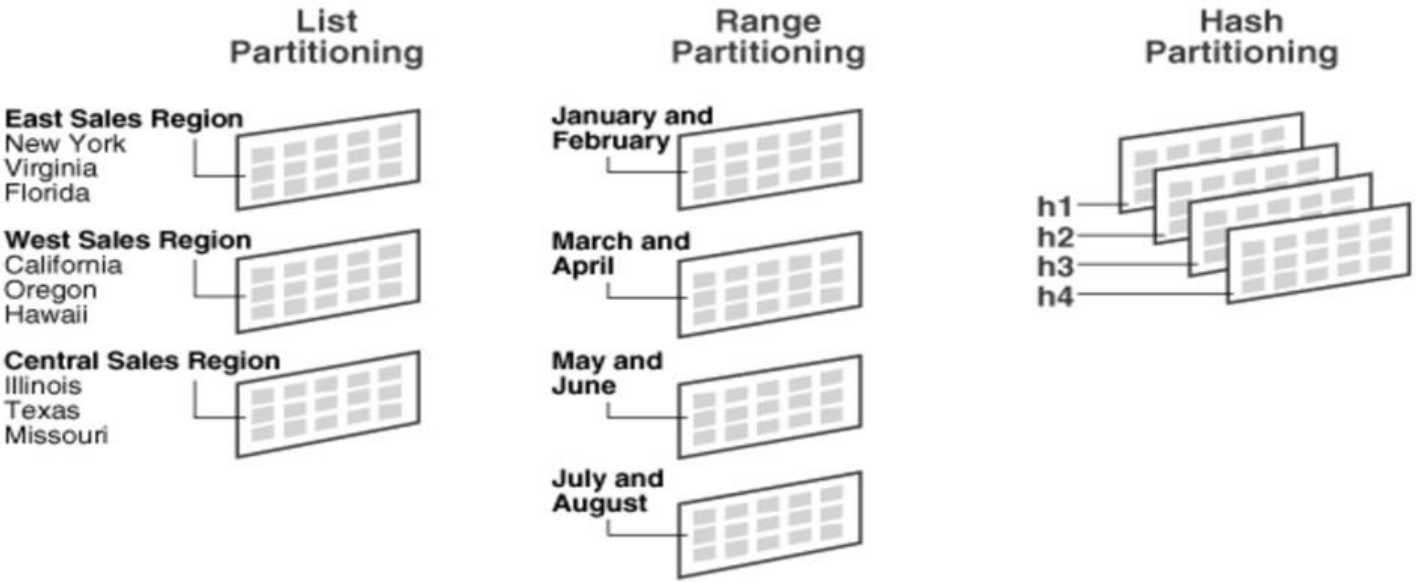
Partitioning can provide several benefits:

- Query performance can be improved dramatically in certain situations, particularly when most of the heavily accessed rows of the table are in a single partition or a small number of partitions. The partitioning substitutes for leading columns of indexes, reducing index size and making it more likely that the heavily-used parts of the indexes fit in memory.
- When queries or updates access a large percentage of a single partition, performance can be improved by taking advantage of sequential scan of that partition instead of using an index and random access reads scattered across the whole table.
- Bulk loads and deletes can be accomplished by adding or removing partitions, if that requirement is planned into the partitioning design. ALTER TABLE and DROP TABLE are both far faster than a bulk DELETE operation
- Seldom-used data can be migrated to cheaper and slower storage media.

SQL – Advanced Objects - Partitioning

PostgreSQL offers built-in support for the following forms of partitioning:

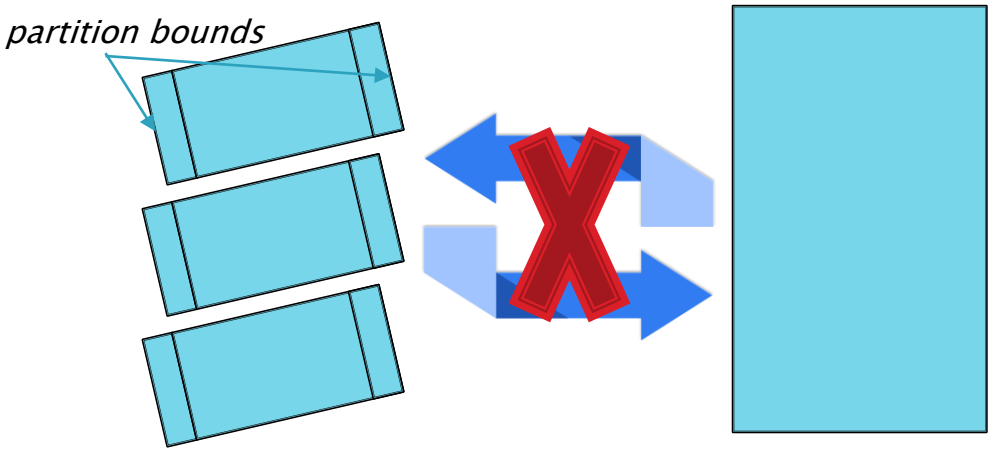
- List Partitioning - The table is partitioned by explicitly listing which key values appear in each partition.
- Range Partitioning - The table is partitioned into "ranges" defined by a key column or set of columns, with no overlap between the ranges of values assigned to different partitions.
- Hash Partitioning - The table is partitioned by specifying a modulus and a remainder for each partition. Each partition will hold the rows for which the hash value of the partition key divided by the specified modulus will produce the specified remainder.



The specification of a partioned table consists of

- The partitioning method
- A list of columns or expressions to be used as the partition key

Partitions may themselves be defined as partitioned tables, using what is called sub-partitioning. Partitions may have their own indexes, constraints and default values, distinct from those of other partitions.



Partition Creation

```
CREATE TABLE measurement (
    logdate         date not null,
    peaktemp        int,
    humidity     int
) PARTITION BY RANGE (logdate);
```

```
CREATE TABLE measurement_y2018m1 PARTITION OF measurement
FOR VALUES FROM ('2018-01-01') TO ('2018-02-01');

CREATE TABLE measurement_y2018m2 PARTITION OF measurement
FOR VALUES FROM ('2018-02-01') TO ('2018-03-01');
... ... ...

CREATE TABLE measurement_y2019m1 PARTITION OF measurement
FOR VALUES FROM ('2019-01-01') TO ('2019-02-01');
```

```
INSERT INTO measurement (logdate, peaktemp, humidity)
        VALUES ('2018-02-10', 24, 52);
```

This insertion goes into measurement_y2018m2 table

Partition Maintenance

Indexes:

```
CREATE INDEX ON measurement (logdate);
```

This automatically creates one index on each partition, and any partitions you create or attach later will also contain the index.

Limitations:

- Primary keys are supported on partitioned tables.
- Foreign keys referencing partitioned tables are not supported but foreign key references from a partitioned table to some other table are supported.
- BEFORE ROW triggers must be defined on individual partitions, not the partitioned table.

Partition Maintenance

Add a new parition

```
CREATE TABLE measurement_y2008m02 PARTITION OF measurement
    FOR VALUES FROM ('2008-02-01') TO ('2008-03-01')
```

```
CREATE TABLE measurement_y2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS)
  TABLESPACE fasttablespace;

ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );

ALTER TABLE measurement ATTACH PARTITION measurement_y2008m02
    FOR VALUES FROM ('2008-02-01') TO ('2008-03-01' );
```

Delete a partition

```
ALTER TABLE measurement DETACH PARTITION measurement_y2006m02;

DROP TABLE measurement_y2006m02;
```

Common SQL Programming Mistakes

1. Forgotten Primary Keys
2. Poorly Managed Data Redundancy
3. Avoid NOT IN or IN and Use JOIN Instead
4. Forgotten NULL vs. Empty String Values
5. The Asterisk Character in SELECT Statements
6. Data Mismatches in Column Assignment
7. Logical OR and AND Operations
8. JOIN on Indexes

```
SELECT questions FROM audience;
```

SQL – Advanced Objects – SQL Architecture