Database Tuning

Model

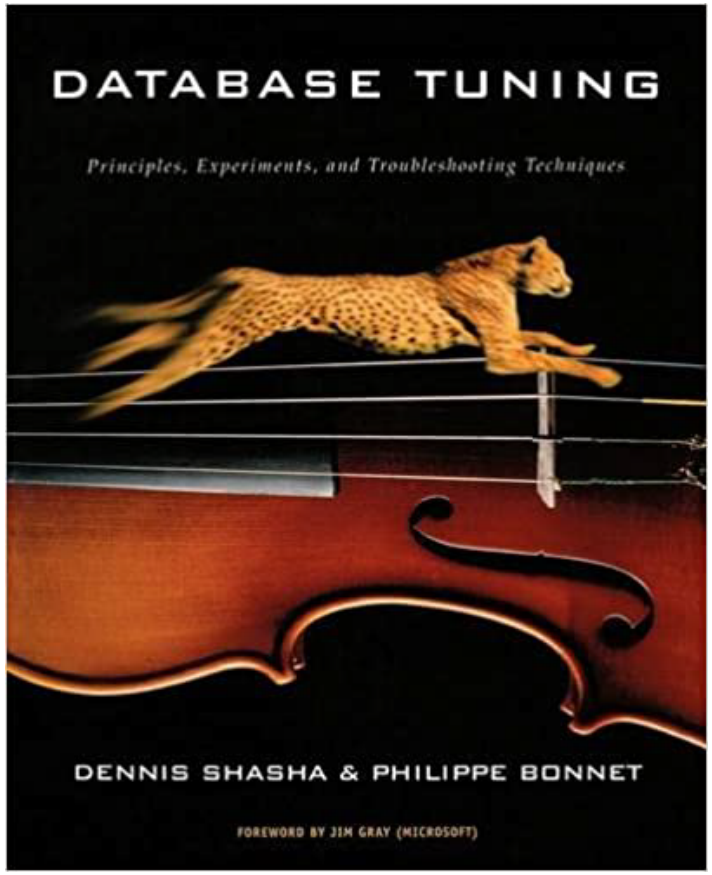
SQL Tuning

Book | TPC-C | PostgreSQL | Explain | Analyze

Database Tuning

Database Tuning by Dennis Shasha and Philippe Bonnet

> *Tuning rests on a foundation of informed common sense. This makes it both easy and hard. [...] Tuning is easy because the tuner needs not struggle through complicated formulas or theorems. [...] Tuning is difficult because the principles and knowledge underlying the common sense require a broad and deep understanding [...].*

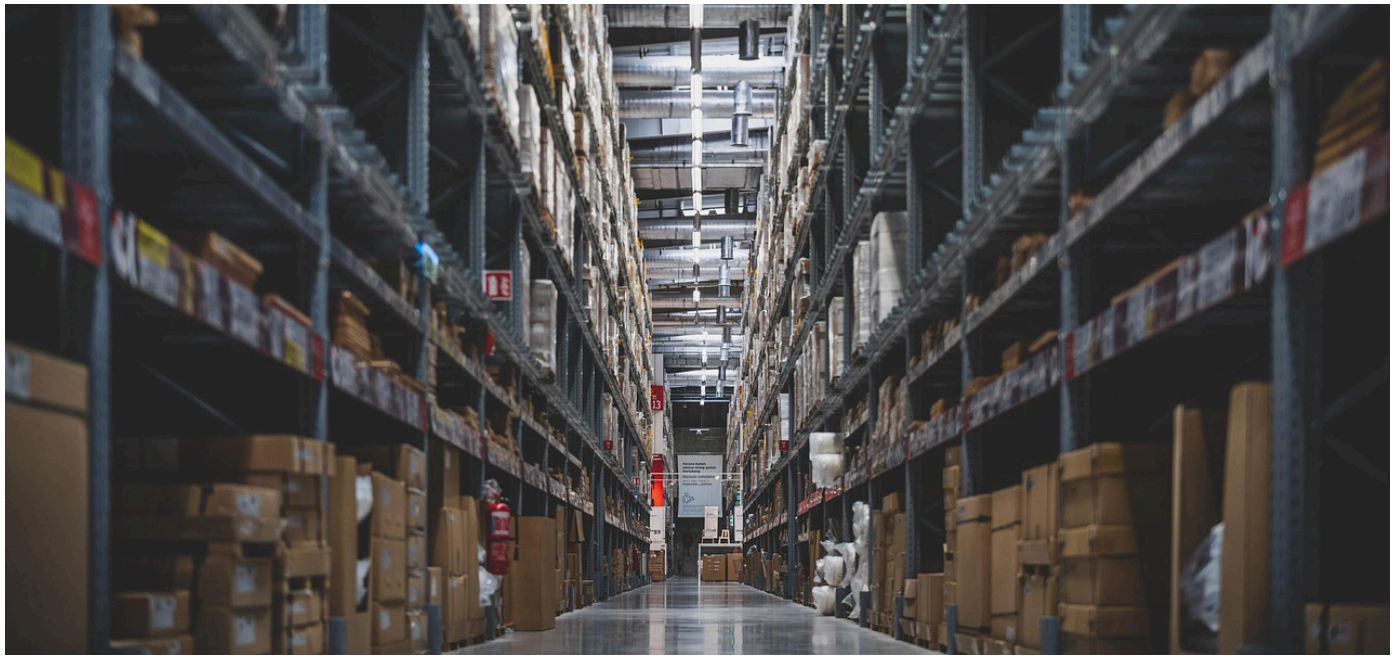


Partial Schema

The TPC-C Schema (partial)

A wholesale supplier operates out of several warehouses. The warehouse maintain stocks for the items sold by the company. We record the quantity in stock for each item available in each warehouse.

Warehouses have a unique identifier, a name, and a location. A location is given by a street, city, and country. Items have a unique identifier, a unique image identifier, a name, and a price.

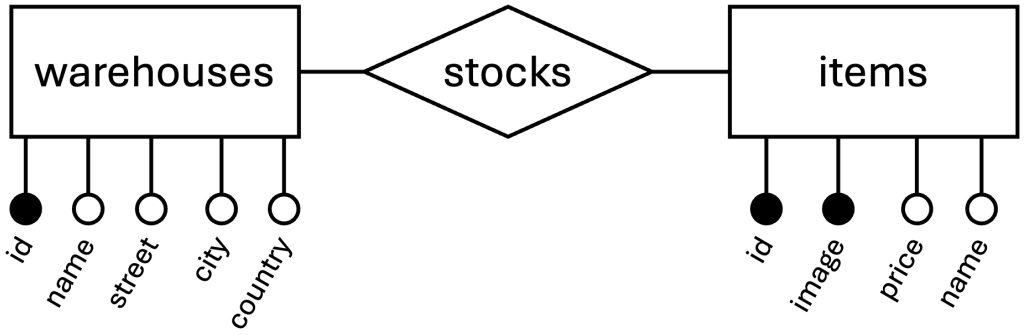


Entity-Relationship Diagram

The TPC-C Schema (partial)

A wholesale supplier operates out of several warehouses. The warehouse maintain stocks for the items sold by the company. We record the quantity in stock for each item available in each warehouse.

Warehouses have a unique identifier, a name, and a location. A location is given by a street, city, and country. Items have a unique identifier, a unique image identifier, a name, and a price.



Book | TPC-C ❶ ❷ ❸ ❹ ❺ | PostgreSQL | Explain | Analyze

Items

The TPC-C Schema (partial)

A wholesale supplier operates out of several warehouses. The warehouse maintain stocks for the items sold by the company. We record the quantity in stock for each item available in each warehouse.

Warehouses have a unique identifier, a name, and a location. A location is given by a street, city, and country. Items have a ***unique identifier***, a ***unique image identifier***, a ***name***, and a ***price***.

Schema

```
CREATE TABLE items (
  i_id INTEGER PRIMARY KEY,
  i_im_id CHAR(8) UNIQUE NOT NULL,
  i_name VARCHAR(64) NOT NULL,
  i_price NUMERIC NOT NULL
    CHECK (i_price > 0)
);
```

Query

```
SELECT * FROM items;
```

Items

i_id	i_im_id	i_name	i_price
1	35356226	Indapamide	95.23
2	00851287	SYLATRON	80.22
3	52549414	Meprobamate	11.64
:	:	:	:

483 rows

Warehouses

The TPC-C Schema (partial)

A wholesale supplier operates out of several warehouses. The warehouse maintain stocks for the items sold by the company. We record the quantity in stock for each item available in each warehouse.

Warehouses have a ***unique identifier***, a ***name***, and a ***location***. A location is given by a ***street***, ***city***, and ***country***. Items have a unique identifier, a unique image identifier, a name, and a price.

Schema

```
CREATE TABLE warehouses (
  w_id INTEGER PRIMARY KEY,
  w_name VARCHAR(16) NOT NULL,
  w_street VARCHAR(32) NOT NULL,
  w_city VARCHAR(32) NOT NULL,
  w_country VARCHAR(16) NOT NULL
);
```

Query

```
SELECT * FROM warehouses;
```

Warehouses

w_id	w_name	w_street	w_city	w_country
301	Schmedeman	Sunbrook	Singapore	Singapore
281	Crescent Oaks	Loeprich	Singapore	Singapore
22	Namekagon	Anniversary	Singapore	Singapore
:	:	:	:	:

1005 rows

Book | TPC-C ❶ ❷ ❸ ❹ ❺ | PostgreSQL | Explain | Analyze

Stocks

The TPC-C Schema (partial)

A wholesale supplier operates out of several warehouses. The warehouse maintain **stocks** for the items sold by the company. We record the ***quantity*** in stock **for each item available in each warehouse***.

Warehouses have a unique identifier, a name, and a location. A location is given by a street, city, and country. Items have a unique identifier, a unique image identifier, a name, and a price.

Schema

```
CREATE TABLE stocks (
  w_id INTEGER REFERENCES warehouses(w_id),
  i_id INTEGER REFERENCES items(i_id),
  s_qty SMALLINT
    CHECK (s_qty > 0),
  PRIMARY KEY (w_id, i_id)
);
```

Query

```
SELECT * FROM stocks;
```

Stocks

w_id	i_id	s_qty
301	1	338
301	4	938
301	5	760
:	:	:

44912 rows

*If an item is not available in a warehouse, then there is no entry for this pair.

Query

Question

How many units of Aspirin are in stock in Indonesia?

SQL

```
SELECT SUM(s.s_qty)
FROM items i
  NATURAL JOIN stocks s
  NATURAL JOIN warehouses w
WHERE i.i_name = 'Aspirin'
  AND w.w_country = 'Indonesia';
```

Result

sum
35599

1 rows

Behind the Scene?

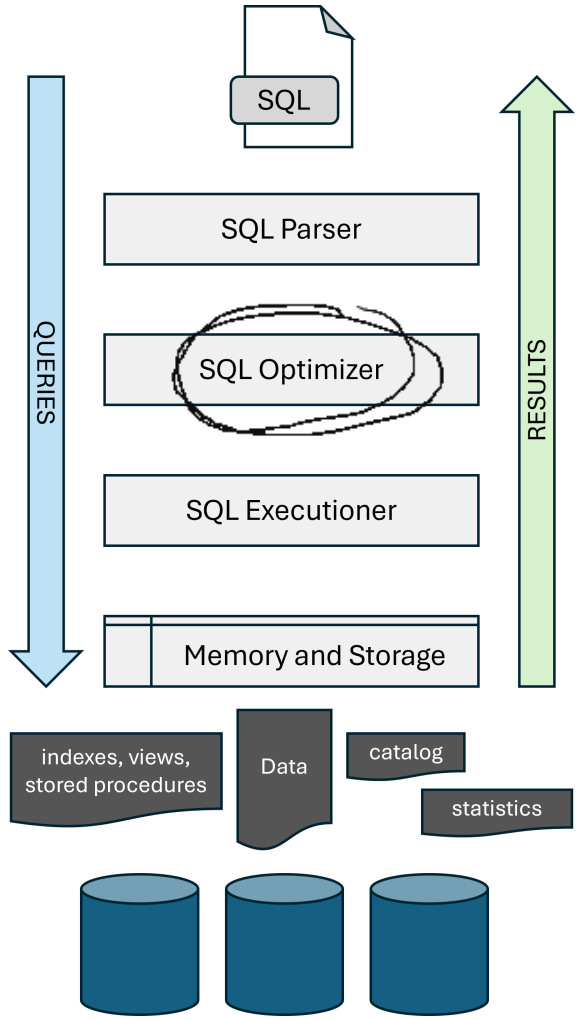
What is happening behind the scene when PostgreSQL is answering the query?

Architecture

PostgreSQL Architecture

PostgreSQL architecture consists of the following components.

- **SQL Parser** and **Rewriter**
  Take the SQL query and convert it into a parse tree ready for optimization.
- **Planner**/**Optimizer**
  Generates a query execution plan.
- **Execution Engine**
  Executes the plan.
- **Memory and Storgage Management Systems**
  Manages memory.



Optimizer

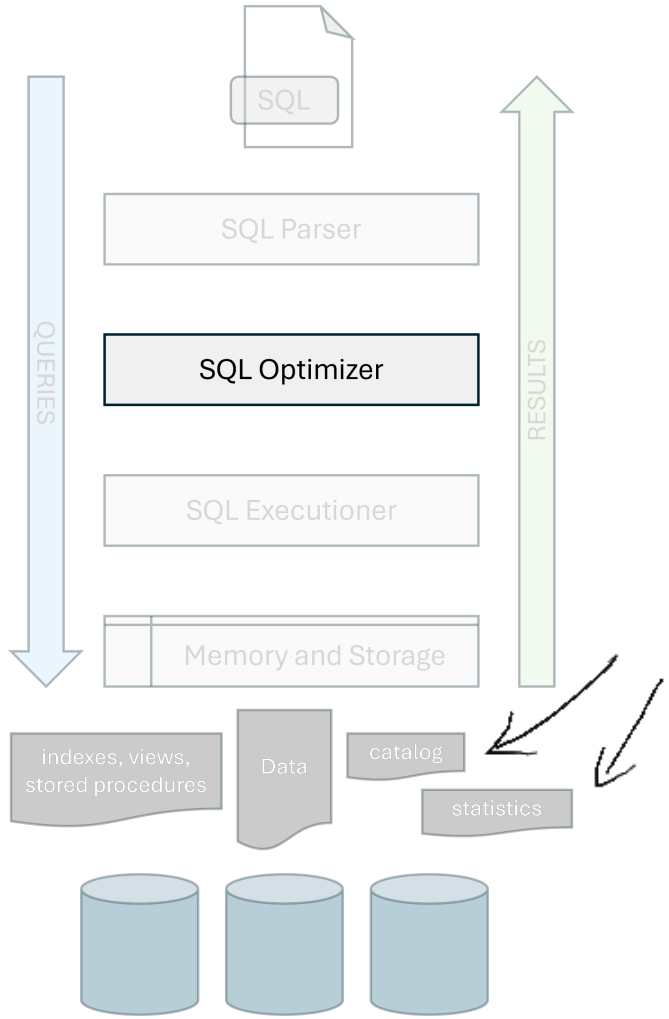
Planner/Optimizer

The **planner/optimizer** generates a query execution plan. It uses catalogue and statistics to devise candidate plans and estimate their respective costs. It chooses the plan with the ***least estimated cost***.

Execution Plan

An **execution plan** is a directed acyclic graph or a tree of physical ***algebraic operators***. In PostgreSQL it includes the following.

- Sequential/Index Scans
- Sorting
- Aggregation Operators
- Nested Loops
- Hash/Merge Joins

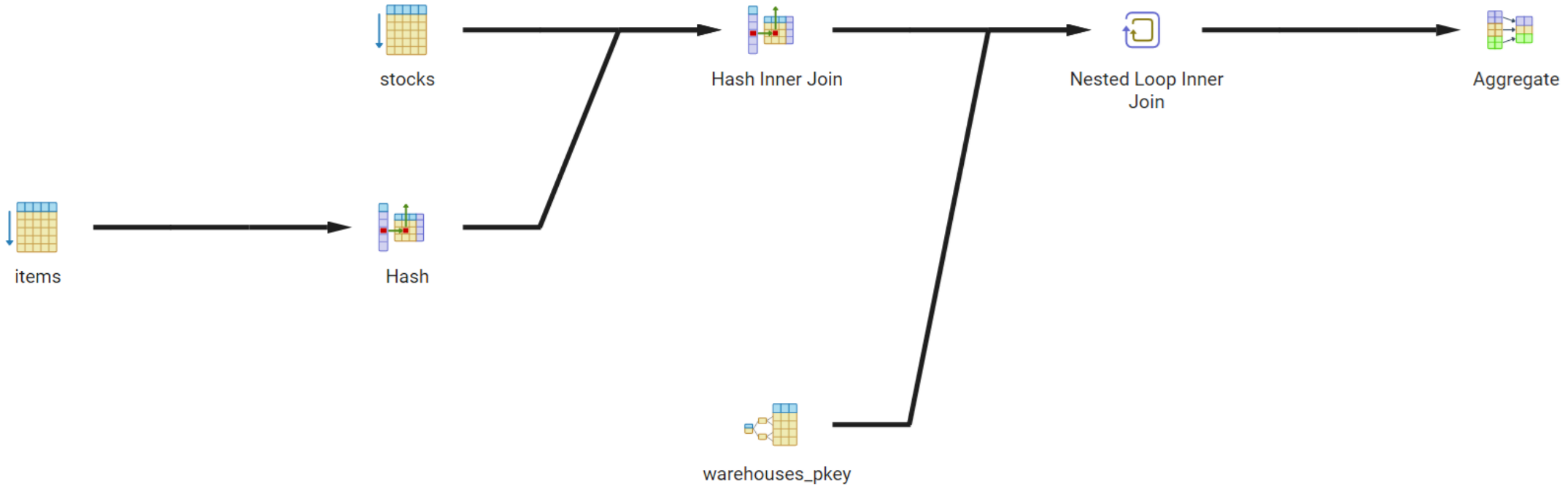


Book | TPC-C | PostgreSQL 1 2 3 4 5 | Explain | Analyze

Execution

Query

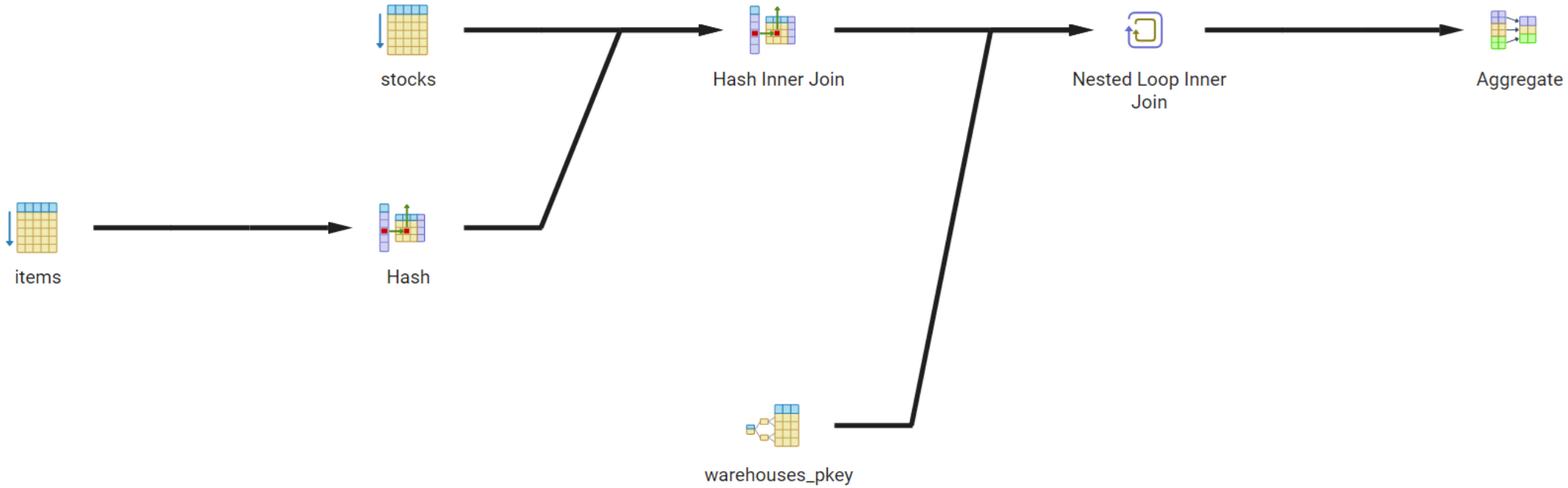
```
SELECT SUM(s.s_qty)
FROM items i NATURAL JOIN stocks s NATURAL JOIN warehouses w
WHERE i.i_name = 'Aspirin' AND w.w_country = 'Indonesia';
```



Execution

Engine

The **execution engine** executes the plan, accessing the data, and indexes as well as executing the necessary operators, functions, and procedures.



Book | TPC-C | PostgreSQL 1 2 3 4 5 | Explain | Analyze

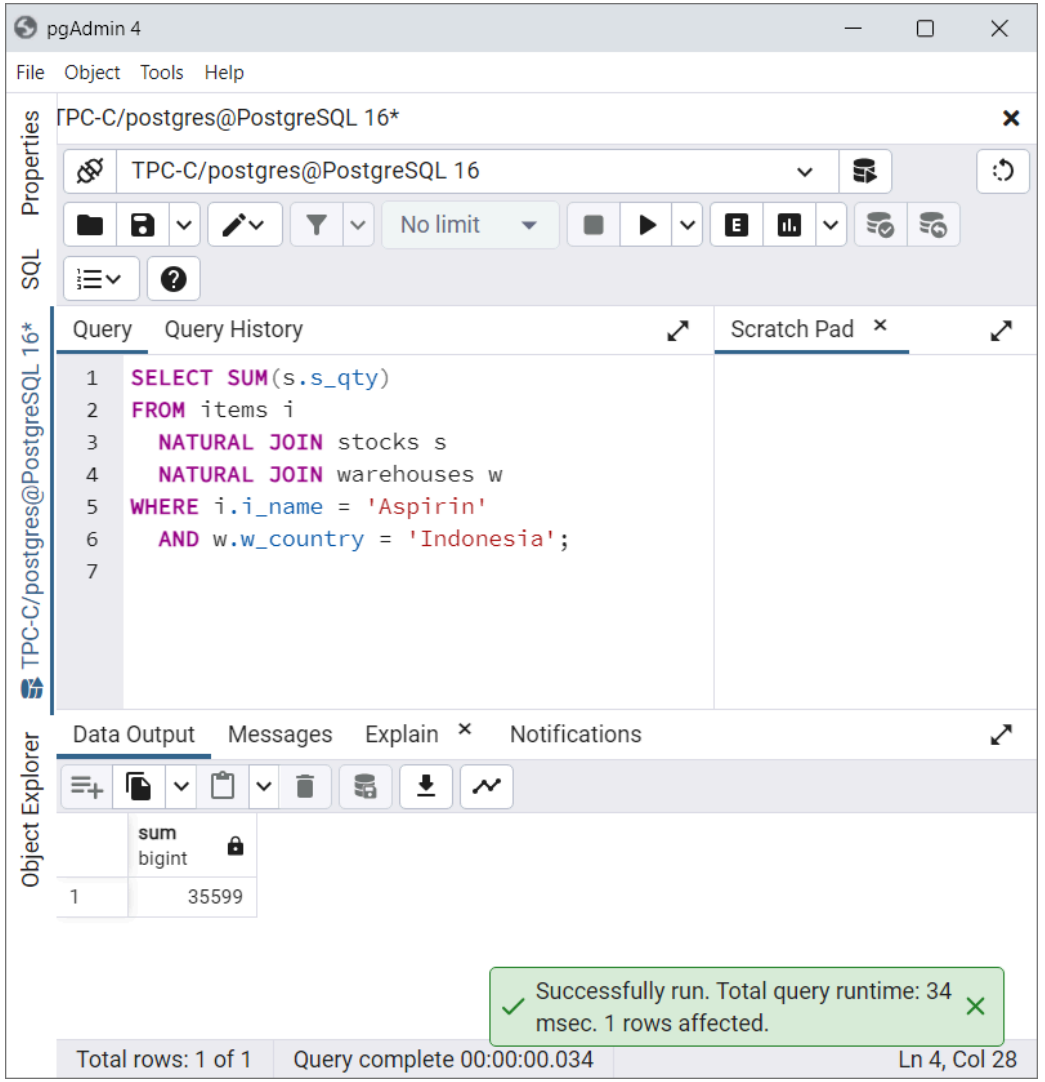
Execution

Time

The **total query time** indicated by pgAdmin4 includes **query planning time**, **execution time**, **network latency**, and **client overhead**.

{ planning + Executing }

1 min 2 s



Explain Command

The EXPLAIN Command

The command `EXPLAIN` generates and displays the ***annotated execution plan*** that the PostgreSQL planner generates for the supplied statement.

The command `EXPLAIN` with the (`VERBOSE`) option displays the annotated execution plan with ***more details***.

Supported SQL Constructs

The command `EXPLAIN` only supports the following SQL constructs: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `EXECUTE` *(of a prepared statement)*, `CREATE TABLE`, and `DECLARE` *(of a cursor)*.

Explain and Analyze

The command `EXPLAIN` with the (`ANALYZE`) option executes the query and shows the ***execution time***, as well as ***row count*** for each step.

Explain Command

Query

```
EXPLAIN SELECT SUM(s.s_qty)
FROM items i NATURAL JOIN stocks s NATURAL JOIN warehouses w
WHERE i.i_name = 'Aspirin' AND w.w_country = 'Indonesia';
```

Question

Can you retry with the following?

```
EXPLAIN (VERBOSE)
EXPLAIN (ANALYZE)
```

Query Plan
Aggregate (cost=849.87..849.88 rows=1 width=8)
-> Nested Loop (cost=11.33..849.65 rows=89 width=2)
-> Hash Join (cost=11.05..822.10 rows=93 width=6)
Hash Cond: (s.i_id = i.i_id)
-> Seq Scan on stocks s (cost=0.00..692.12 rows=44912 width=10)
-> Hash (cost=11.04..11.04 rows=1 width=4)
-> Seq Scan on items i (cost=0.00..11.04 rows=1 width=4)
Filter: ((i_name)::text = 'Aspirin'::text)
-> Index Scan using warehouses_pkey on warehouses w (cost=0.28..0.30 rows=1 width=4)
Index Cond: (w_id = s.w_id)
Filter: ((w_country)::text = 'Indonesia'::text)

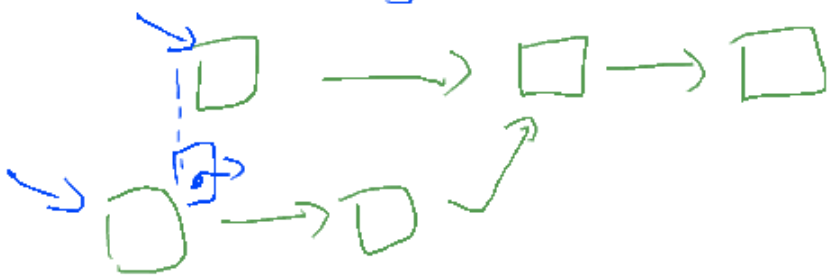
Explain Command

Estimates

At each node of the plan, EXPLAIN gives three estimates: `cost`, `rows`, and `width`.

- `cost` : The cost attribute is the estimated startup cost *(i.e., cost before any output is produced)* and total cost of executing a node *(as well as their children)* with the query execution plan.
  Two costs are expressed in ***arbitrary units*** *(roughly proportional to the estimated time)*. Lower costs generally indicate more efficient plans. The cost include the estimated central processing unit cost and the estimated I/O costs associated with executing the node.
  They do not include the cost of transmission to the client.
- `rows` : The rows attribute is the ***estimated number of rows*** a node returns.
- `width` : The width attribute is the ***estimated average width*** *(measured in bytes)* of the rows produced by the node.

The product of rows and width estimates the ***total data volume*** for the result set.



Estimation Mismatch

Actual Runtime Behavior

The execution plan is the plan executed by the execution engine. The planner/optimizer has used the estimates to select the execution plan for the query.

However, the estimates may **not always match the actual runtime behavior**.

Tuning

By considering the execution plan and the estimates, the programmer gains **insights**

- into how the database engine manipulates data, uses indexes, and performs other operations to execute the statement,
- about the cost estimated, and
- about the information available to the planner/optimizer.

The programmer can look at the execution plan and at the estimates of different queries to ***try to learn how to tune the SQL code*** *(if necessary)*.

Planning

Catalogue and Statistics

PostgreSQL query planner/optimizer uses the catalogue and statistics built *(and maintained)* by PostgreSQL. We can check the system catalogue tables and views.

- `pg_tables` — The view provides access to useful information *(e.g., name, indexes, triggers, etc)* about each table in the database.
- `pg_attribute` — The catalog stores information about table columns. There will be exactly one row for every column in every table in the database.
- `pg_statistics` — The catalog stores statistical data about the contents of the database. Entries are created by `ANALYZE` and subsequently used by the query planner. All statistical data in inherently ***approximate***, even assuming that it is up-to-date.
- `pg_stats` — The view provides access to the information stored in the `pg_statistic` catalog. This view allows access only to rows of `pg_statictic` that corresponds to tables the user has permission to read.

The `ANALYZE` command collects statistics about the contents of the tables in the database and stores the results in the `pg_statistic` system catalogue.

*`ANALYZE` is different from `EXPLAIN` with `ANALYZE`.

Statistic

Query

```
SELECT * FROM pg_stats
WHERE tablename = 'warehouses' AND attname = 'w_city';
```

...	avg_width	most_common_vals	most_common_freqs	...
...	9	{"Kota Kinabalu","Kuala Lumpur","Singapore",...}	{0.0059701493,0.0049751243,0.0049751243,...}	...

pg_stats

For instance, the view `pg_stats` records that `"Kota Kinabalu"` is the most common value of the column `w_city`. The estimated frequency is `0.0059701493`.

The next most common value is `"Kuala Lumpur"` with estimated frequency of `0.0049751243`.

Among other statistics, it also records the average width of columns *(i.e., 9)* in bytes used in the attribute `width` of EXPLAIN.

Explain Analyze

Query

```
EXPLAIN (ANALYZE) SELECT SUM(s.s_qty)
FROM items i NATURAL JOIN stocks s NATURAL JOIN warehouses w
WHERE i.i_name = 'Aspirin' AND w.w_country = 'Indonesia';
```

Query Plan
Aggregate (cost=849.87..849.88 rows=1 width=8) (actual time=4.122..4.125 rows=1 loops=1)
-> Nested Loop (cost=11.33..849.65 rows=89 width=2) (actual time=0.138..4.113 rows=72 loops=1)
-> Hash Join (cost=11.05..822.10 rows=93 width=6) (actual time=0.079..3.978 rows=76 loops=1)
:
-> Hash (cost=11.04..11.04 rows=1 width=4) (actual time=0.033..0.034 rows=1 loops=1)
Buckets: 1024 Batches: 1 Memory Usage: 9kB
-> Seq Scan on items i (cost=0.00..11.04 rows=1 width=4) (actual time=0.023..0.031 rows=1 loops=1)
:
Planning Time: 2.326 ms
Execution Time: 4.156 ms

Explain Analyze

Measurements

At each node of the plan, `EXPLAIN` gives four measurements: `actual time`, `rows`, `loops`, and `Rows Removed by Filter`.

- `actual time` The actual time attribute is the average startup and total execution time of the node in *milliseconds*.
- `rows` The rows attribute is the *actual number of rows* a node returns.
- `loops` The loops attribute is the *number of times* the node *(i.e., the subplan)* is executed.
- `Rows Removed by Filter` The rows removed by filter attribute is the *number of rows filtered out* by the condition.

The product of actual time and the loops is the *total time of a node*.

Explain Analyze

Additional Information

`EXPLAIN (ANALYZE)` adds overhead to the overall time.

The execution time **vary** from one execution to the next. It is necessary to ***average*** the measurements.

The costs and actual times are in **different units**. They should be ***approximately commensurate*** if the estimation was good. It is more important to look at whether the ***estimated row counts*** are reasonably close to reality.

Timing Information

`EXPLAIN (ANALYZE)` also gives the **planning time** and **execution time**. The execution time includes execution ***start-up***, ***shut-down*** time, and time spent ***processing the result*** rows.

Explain Analyze

Other Options

Other `EXPLAIN` options include the following.

- `COSTS`
- `SETTINGS`
- `GENERIC_PLAN`
- `BUFFERS`
- `WAL`
- `TIMING`
- `SUMMARY`
- `FORMAT { TEXT | XML | JSON | YAML }`

Performance

To understand performance well, the programmer should run the queries many times and ***look at the average***. Statistics are gathered. Pages are brought to the memory buffer.

The commands `VACUUM`, `ANALYZE`, and `VACUUM ANALYZE` reorganizes data and collect statistics. The costs, the times, and the plans change *(are improved)* accordingly.

Explain Analyze

Query

```
EXPLAIN (ANALYZE, FORMAT JSON) (
  SELECT SUM(s.s_qty)
  FROM items i
    NATURAL JOIN stocks s
    NATURAL JOIN warehouses w
  WHERE i.i_name = 'Aspirin'
    AND w.w_country = 'Indonesia'
);
```


```
[
  {
    "Plan": {
      "Node Type": "Aggregate",
      "Strategy": "Plain",
      "Partial Mode": "Simple",
      "Parallel Aware": false,
      "Async Capable": false,
      "Startup Cost": 849.87,
      "Total Cost": 849.88,
      "Plan Rows": 1,
      "Plan Width": 8,
      "Actual Startup Time": 14.406,
      "Actual Total Time": 14.410,
      "Actual Rows": 1,
      "Actual Loops": 1,
      "Plans": [ ... ]
    },
    "Planning Time": 0.565,
    "Triggers": [ ],
    "Execution Time": 7.308
  }
]
```

Book | TPC-C | PostgreSQL | Explain | Analyze ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽

Visualization

#	Node	Timings		Rows		
		Exclusive	Inclusive	Rows X	Actual	Plan
1.	→ Aggregate (cost=849.87..849.88 rows=1 width=8) (actual=6.531..6.…	0.015 ms	6.533 ms	↑ 1	1	1
2.	→ Nested Loop Inner Join (cost=11.33..849.65 rows=89 width=2) …	0.126 ms	6.519 ms	↑ 1.24	72	89
3.	→ Hash Inner Join (cost=11.05..822.1 rows=93 width=6) (act… **Hash Cond**: (s.i_id = i.i_id)	3.382 ms	6.393 ms	↑ 1.23	76	93
4.	→ Seq Scan on stocks as s (cost=0..692.12 rows=44912…	2.95 ms	2.95 ms	↑ 1	44912	44912
5.	→ Hash (cost=11.04..11.04 rows=1 width=4) (actual=0.0… **Buckets**: 1024 **Batches**: 1 **Memory Usage**: 9 kB	0.004 ms	0.061 ms	↑ 1	1	1
6.	→ Seq Scan on items as i (cost=0..11.04 rows=1 wi… **Filter**: ((i_name)::text = 'Aspirin'::text) **Rows Removed by Filter**: 482	0.057 ms	0.057 ms	↑ 1	1	1

Break

Back by 20:09

Sequential Scan

Scanning

Question

Find the name of the warehouses in the city of Singapore.

Query

```
SELECT w.w_name
FROM warehouses w
WHERE w.w_city = 'Singapore';
```

Result

w_name
Schmedeman
Crescent Oaks
Namekagon
Fairfield
Briar Crest

5 rows

Sequential Scan

Scanning

Question

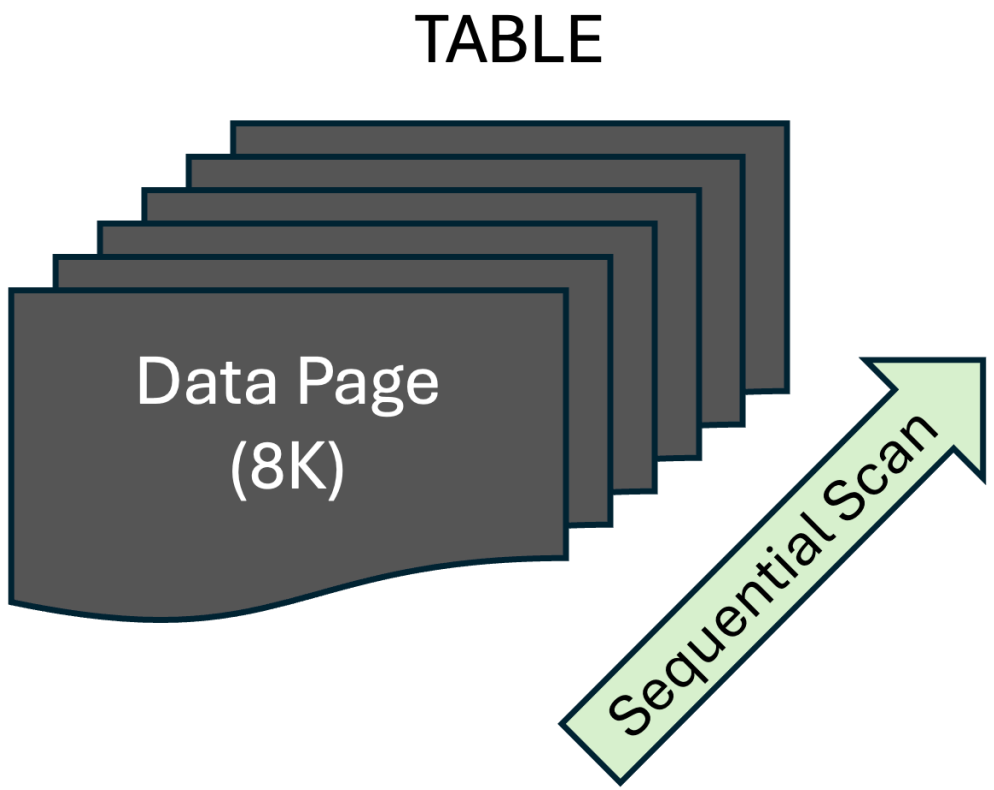
Find the name of the warehouses in the city of Singapore.

Query

```
SELECT w.w_name
FROM warehouses w
WHERE w.w_city = 'Singapore';
```

Scan

If the statistics indicate that the percentage of data to retrieve is large or scattered, and if it is not possible or worth trying to prepare and use another method than a sequential scan, then the optimizer uses a ***sequential scan***.



*PostgreSQL default page size is 8KB.

Sequential Scan

Scanning

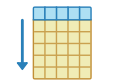
Question

Find the name of the warehouses in the city of Singapore.

Query

```
EXPLAIN SELECT w.w_name
FROM warehouses w
WHERE w.w_city = 'Singapore';
```

Query Plan
Seq Scan on warehouses w (cost=0.00..21.56 rows=5 width=7)
Filter: ((w_city)::text = 'Singapore'::text)



warehouses

Sequential Scan

Scan

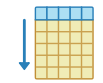
Question

Find the name of the warehouses in the city of Singapore.

Query

```
EXPLAIN (ANALYZE) SELECT w.w_name
FROM warehouses w
WHERE w.w_city = 'Singapore';
```

Query Plan
Seq Scan on warehouses w (cost=0.00..21.56 rows=5 width=7) (actual time=0.056..0.217 rows=5 loops=1)
Filter: ((w_city)::text = 'Singapore'::text)
Rows Removed by Filter: 1000
Planning Time: 0.180 ms
Execution Time: 0.276 ms



warehouses

Sequential Scan

Scanning + Sorting

Question

Find the name of the warehouses in the city of Singapore. Sort the output in ascending order of name.

Query

```
EXPLAIN SELECT w.w_name
FROM warehouses w
WHERE w.w_city = 'Singapore'
ORDER BY w.w_name;
```

Query Plan
Sort (cost=21.62..21.63 rows=5 width=7)
Sort Key: w_name
-> Seq Scan on warehouses w (cost=0.00..21.56 rows=5 width=7)
Filter: ((w_city)::text = 'Singapore'::text)

Sequential Scan

Scanning + Sorting

Question

Find the name of the warehouses in the city of Singapore. Sort the output in ascending order of name.

Query

```
EXPLAIN (ANALYZE) SELECT w.w_name
FROM warehouses w
WHERE w.w_city = 'Singapore'
ORDER BY w.w_name;
```

Query Plan
Sort (cost=21.62..21.63 rows=5 width=7) (actual time=0.164..0.166 rows=5 loops=1)
Sort Key: w_name
Sort Method: quicksort Memory: 25kB
-> Seq Scan on warehouses w (cost=0.00..21.56 rows=5 width=7) (actual time=0.031..0.139 rows=5 loops=1)
:

Sequential Scan

Scanning + Grouping

AAAA BBAAA

(A, 4) (B, 2) (A, 3)

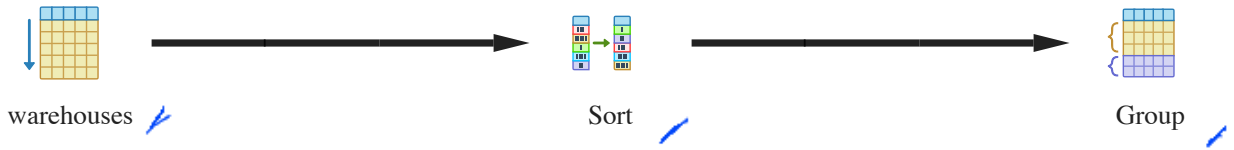
Question

Find the **different** name of the warehouses in the city of Singapore.

Query

```
EXPLAIN SELECT w.w_name
FROM warehouses w
WHERE w.w_city = 'Singapore'
GROUP BY w.w_name;
```

Query Plan
Group (cost=21.62..21.65 rows=5 width=7)
Group Key: w_name
-> Sort (cost=21.62..21.63 rows=5 width=7)
Sort Key: w_name
-> Seq Scan on warehouses w (cost=0.00..21.56 rows=5 width=7)
Filter: ((w_city)::text = 'Singapore'::text)



Sequential Scan

1 2 3 4 5 6 7 8 9 | Index | Index Scan | Multi-Column

Scanning + Distinct

Question

Find the **different** name of the warehouses in the city of Singapore.

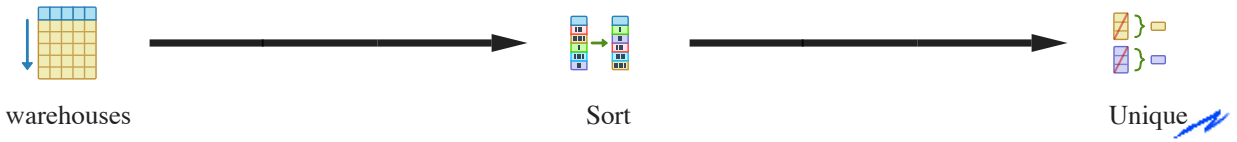
Query

```
EXPLAIN SELECT DISTINCT w.w_name
FROM warehouses w
WHERE w.w_city = 'Singapore';
```

Follow Up Question

Which one is faster?

Query Plan
Unique (cost=21.62..21.65 rows=5 width=7)
Group Key: w_name
-> Sort (cost=21.62..21.63 rows=5 width=7)
Sort Key: w_name
-> Seq Scan on warehouses w (cost=0.00..21.56 rows=5 width=7)
Filter: ((w_city)::text = 'Singapore'::text)



Sequential Scan ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ ❾ | Index | Index Scan | Multi-Column

Scanning + Distinct

Question

Find the different *id* of the warehouses in the city of Singapore.

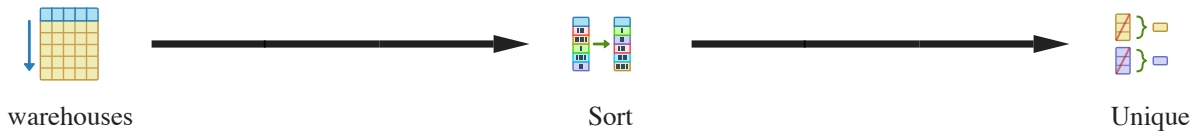
Query

```
EXPLAIN SELECT DISTINCT w.w_id
FROM warehouses w
WHERE w.w_city = 'Singapore';
```

Query Plan
Unique (cost=21.62..21.65 rows=5 width=4)
Group Key: w_name
-> Sort (cost=21.62..21.63 rows=5 width=4)
Sort Key: w_name
-> Seq Scan on warehouses w (cost=0.00..21.56 rows=5 width=7)
Filter: ((w_city)::text = 'Singapore'::text)

Follow Up Question

Is this a good optimization?

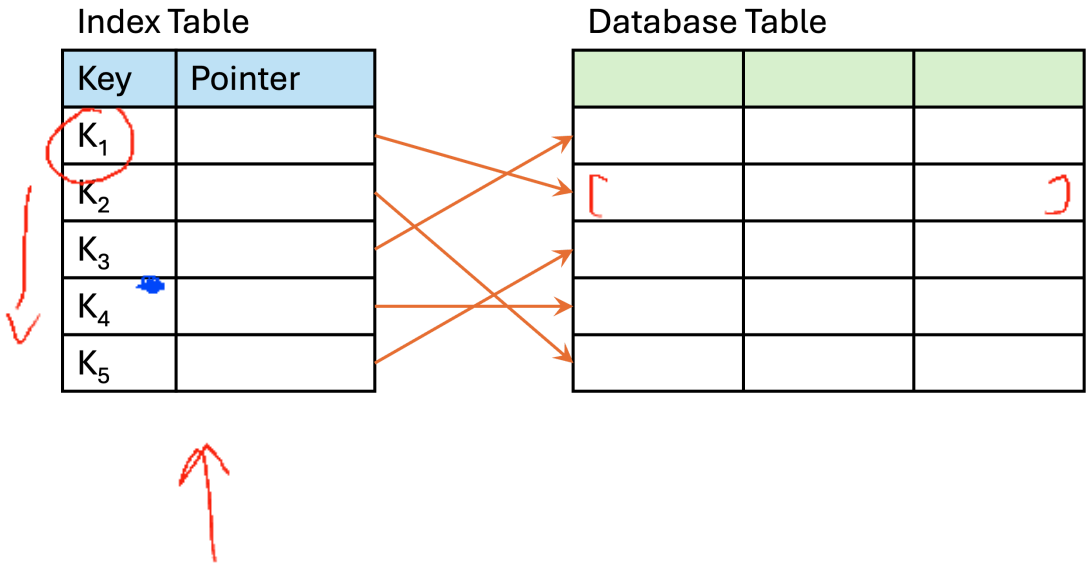


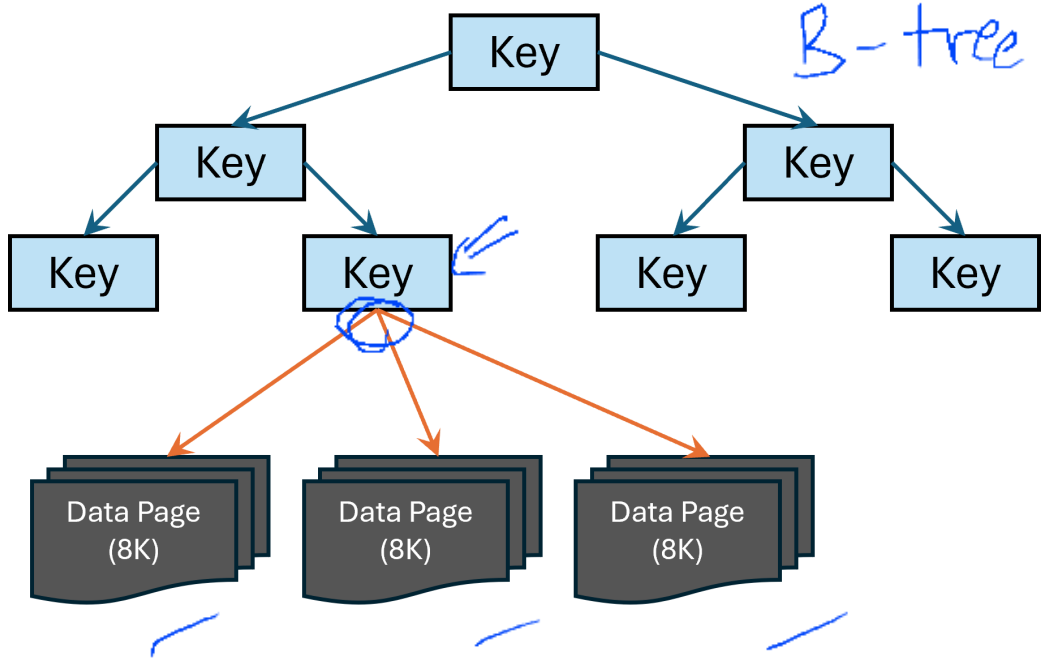
Database Index

Index

An **index** is a data structure that **guides access** to the data.

An index **may or may not speed up** queries, deletions, and updates. It generally slows down insertions and updates *(since both the data and the index must be updated and possibly re-organized)*.





PostgreSQL

PostgreSQL Index

PostgreSQL supports **B-tree** *(default)*, **Hash**, **Generalized Search Tree** *(GiST)*, **Space-Partitioned Generalized Search Tree** *(SP-GiST)*, **Generalized Inverted** *(GIN)*, and **Block Range** *(BRIN)* indices.

Secondary Index

In PostgreSQL, indexes are **secondary**. They are stored ***independently from the table***.

Generally, when searching the index, data must be fetched from both the index and the table *(the heap)*. Although index entries that meet a certain condition are generally located near each other, the corresponding table rows can be situated anywhere.

Scan

Covering Index

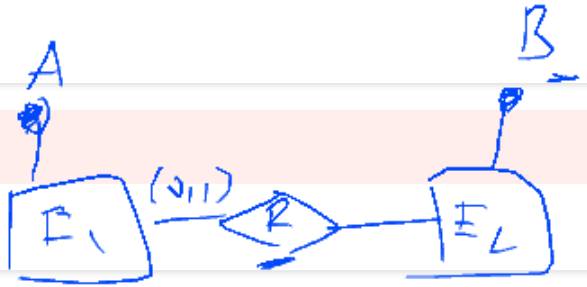
PostgreSQL supports **index-only scans**, which answer a query from the index alone. Index-only scans work for B-tree indexes *(and sometimes other indexes)* when the query references only columns stored in the index *(i.e., when the index is a covering index)*.

Automatic and Non-Automatic

Candidate Keys

PostgreSQL automatically creates a **B-tree unique index** for each `UNIQUE` and `PRIMARY KEY` constraint. The index enforces ***uniqueness*** *(at extra cost for insertions and updates)*.

The index is a **covering index*** for the unique or prime attributes.

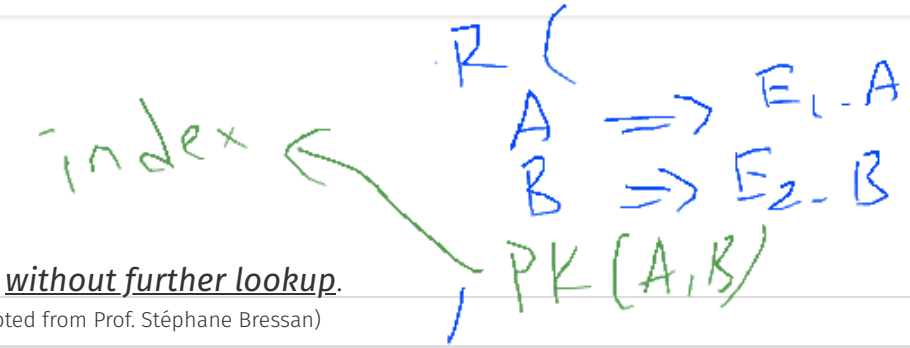


Foreign Keys

PostgreSQL does **not** create an index for `FOREIGN KEY` constraints.

Design Choice

It is up to the designer to decide whether to create an index on the referencing columns and what index to create. Insertion and updates of the referenced table require a scan of the referencing table. Creating an index on the referencing columns may be a good idea. However, foreign key attributes are generally components of a *(composite)* key and are therefore indexed as such.



*A **covering index** is an index that can satisfy all the requested columns in a query ***without further lookup***.

Catalogue

Index Information

The catalogue table and view `pg_index` and `pg_indexes` store information about the indexes in the database.

We can create a view to gather information about the indexes from the catalogue tables and views.

```
CREATE VIEW indexinfo AS
SELECT t.relname AS table_name,
       ix.relname AS index_name,
       i.indisunique AS is_unique,
       i.indisprimary AS is_primary,
       regexp_replace(pg_get_indexdef(i.indexrelid), '.*\((.*)\)', '\1') AS column_names
FROM pg_index i, pg_class t, pg_class ix
WHERE t.oid = i.indrelid AND ix.oid = i.indexrelid;
```

Checking Index

```
SELECT * FROM indexinfo i WHERE i.table_name='warehouses';
```

table_name	index_name	is_unique	is_primary	column_names
warehouses	warehouse_pkey	true	true	w_id

```
SELECT * FROM indexinfo i WHERE i.table_name='items';
```

table_name	index_name	is_unique	is_primary	column_names
items	items_pkey	true	true	i_id
items	items_i_im_id_key	true	false	i_im_id

```
SELECT * FROM indexinfo i WHERE i.table_name='stocks';
```

table_name	index_name	is_unique	is_primary	column_names
stocks	stocks_pkey	true	true	w_id, i_id

Creating Index

CREATE INDEX

We can **create an index** using the `CREATE INDEX` command.

```
CREATE [ UNIQUE ] INDEX [ name ] ON table_name
[ USING method ]
( { column_name | ( expression ) } )
[ WHERE predicate ]
```

- `UNIQUE` checks for duplicate values.
- `method` can be `btree` *(default)*, `hash`, `gist`, or other index types.
- `predicate` defines a ***partial index***.

Creating Index

Example

```
CREATE INDEX items_i_price ON items(i_price);
```

```
SELECT * FROM indexinfo i WHERE i.table_name = 'items';
```

table_name	index_name	is_unique	is_primary	column_names
items	items_pkey	true	true	i_id
items	items_i_im_id_key	true	false	i_im_id
items	items_i_price	false	false	i_price

Scanning

Question

Find the name of the warehouse with identifier 123.

SQL

```
SELECT w.w_name
FROM warehouses w
WHERE w.w_id = '123';
```

Result

w_name
Janyx

1 rows

Difference?

What is the difference between this and checking for `w.w_city = 'Singapore'`?

Scanning with Index

Question

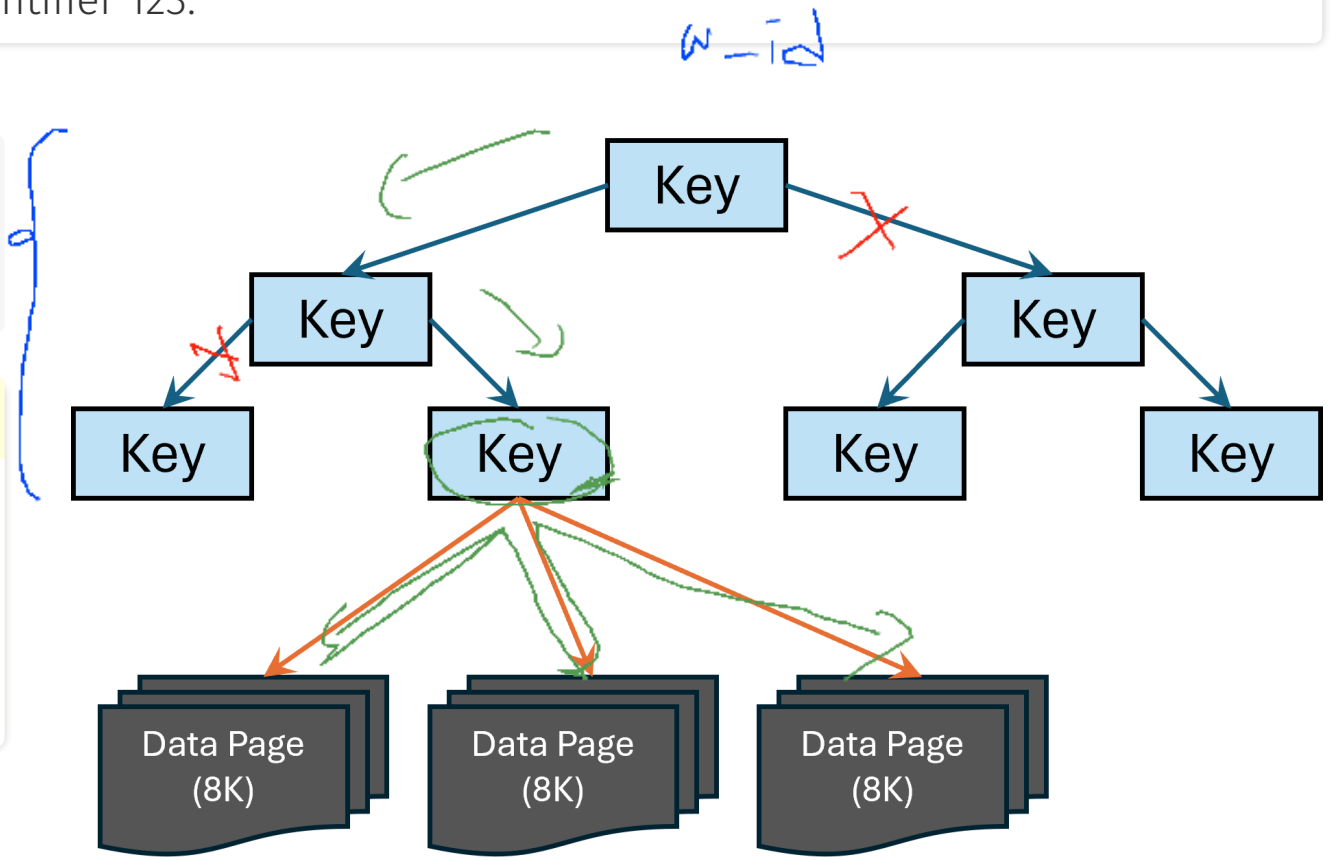
Find the name of the warehouse with identifier 123.

SQL

```
SELECT w.w_name
FROM warehouses w
WHERE w.w_id = '123';
```

Index Scan

If the statistics indicate that the percentage of data to retrieve is **tiny** and if **an index is available**, it may provide ***direct access***. The optimizer uses an ***index scan***.



Scanning with Index

Question

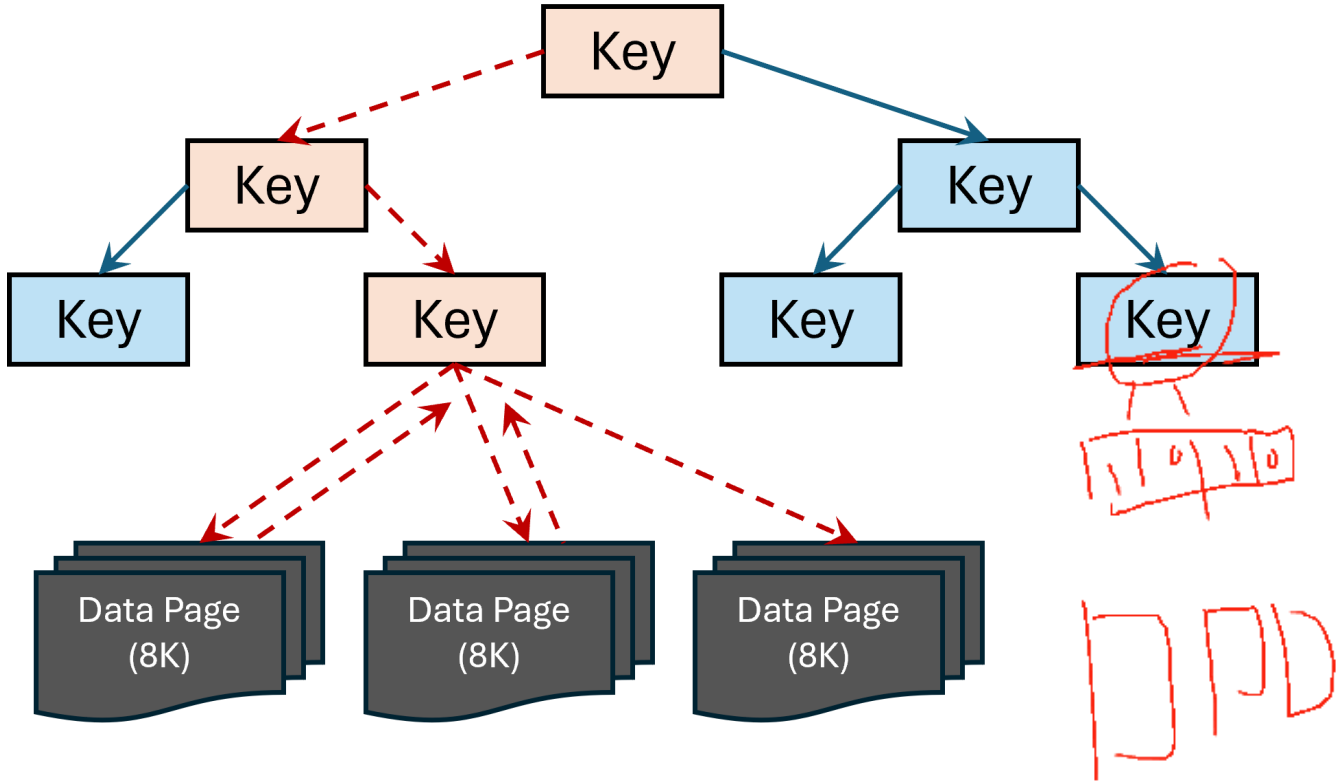
Find the name of the warehouse with identifier 123.

SQL

```
SELECT w.w_name
FROM warehouses w
WHERE w.w_id = '123';
```

Index Scan

If the statistics indicate that the percentage of data to retrieve is **tiny** and if **an index is available**, it may provide ***direct access***. The optimizer uses an ***index scan***.



*Randomly access the index and the table alternately.

Scanning with Index

Question

Find the name of the warehouse with identifier 123.

SQL

```
EXPLAIN SELECT w.w_name
FROM warehouses w
WHERE w.w_id = '123';
```

Query Plan

```
Index Scan using warehouses_pkey on warehouses w
(cost=0.28..8.29 rows=1 width=7)
  Index Cond: (w_id = 123)
```

warehouses_pkey

Scanning with (Additional) Index

Creating an Index

Create a B-tree index *(default)* on the `w_city` attribute of `warehouses`.

SQL

```
CREATE INDEX warehouses_w_city ON warehouses(w_city);
```

```
SELECT * FROM indexinfo i WHERE i.table_name = 'warehouses';
```

table_name	index_name	is_unique	is_primary	column_names
warehouses	warehouses_pkey	true	true	w_id
warehouses	warehouses_w_city	false	false	w_city

Scanning with Bitmap Index

Question

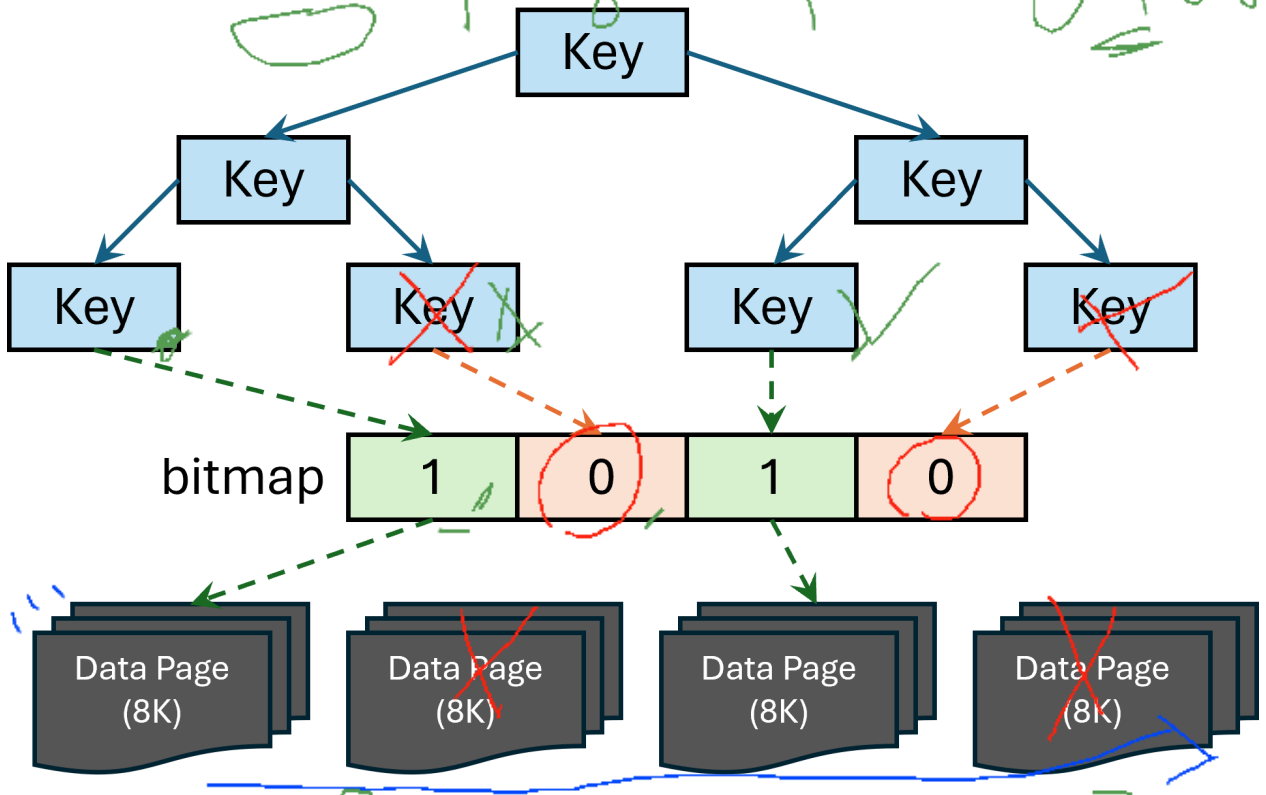
Find the name of the warehouse in the city of Singapore.

SQL

```
EXPLAIN SELECT w.w_name
FROM warehouses w
WHERE w.w_city = 'Singapore';
```

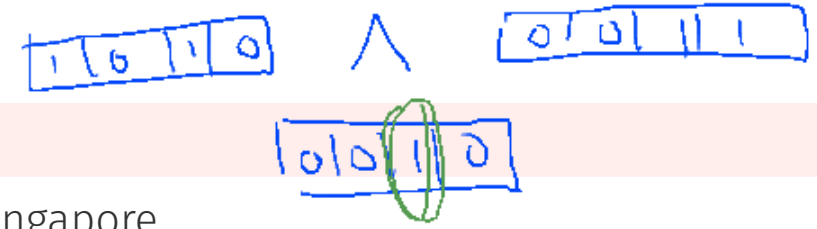
Bitmap Index Scan

If the statistics indicate that the percentage of data to retrieve is **average** and if **an index is available**, a bitmap built on the index may provide *somehow direct access*. The optimizer uses a *bitmap heap scan*.



*The bitmap is an **in-memory** data structure that encodes the pages containing the row. You may need `AND` and `OR` operations on the bitmap.

Scanning with Bitmap Index



Question

Find the name of the warehouse in the city of Singapore.

SQL

```
EXPLAIN SELECT w.w_name
FROM warehouses w
WHERE w.w_city = 'Singapore';
```

Query Plan
Bitmap Heap Scan on warehouses w (cost=4.31..12.38 rows=5 width=37)
Recheck Cond: ((w_city)::text = 'Singapore'::text)
-> Bitmap Index Scan on warehouses_w_city (cost=0.00..4.31 rows=5 width=0)
Index Cond: ((w_city)::text = 'Singapore'::text)

Note

Bitmap Index Scan is followed by Bitmap Heap Scan in PostgreSQL.

Clustering

Table Clustering

We can **cluster the table** usign the index. This would need to be done regularly *(if there are updates)*. PostgreSQL does not dynamically maintain the clustered table!

SQL

```
EXPLAIN (ANALYZE) SELECT w.w_name
FROM warehouses w
WHERE w.w_city = 'Singapore';
```

```
SELECT * FROM warehouses;
```

Cluster

```
CLUSTER warehouses USING warehouses_w_city;
```

Cluster

Physically reorder the rows of underlying table to coincide with the key order in the given index.

A table can only have one cluster index.

Index Scan with Multiple Column

Question

Find the quantity of items with id 7 on warehouse with id 123.

SQL

```
EXPLAIN SELECT s.s_qty
FROM stocks s
WHERE s.w_id = '123' AND s.i_id = '7';
```

Query Plan

```
Index Scan using stocks_pkey on stocks s
(cost=0.29..8.31 rows=1 width=2)
  Index Cond: ((w_id = 123) AND (i_id = 7))
```

MultiColumn Index

A multicolumn index can be used for index scan.

stocks_pkey

Partial Condition

Question

Find the quantity of items on warehouse with id 123.

SQL

```
EXPLAIN SELECT s.s_qty
FROM stocks s
WHERE s.w_id = '123'                    ;
```

Query Plan
Bitmap Heap Scan on stocks s (cost=5.53..235.38 rows=160 width=2)
Recheck Cond: (w_id = 123)
-> Bitmap Index Scan on stocks_pkey (cost=0.00..5.49 rows=160 width=0)
Index Cond: (w_id = 123)

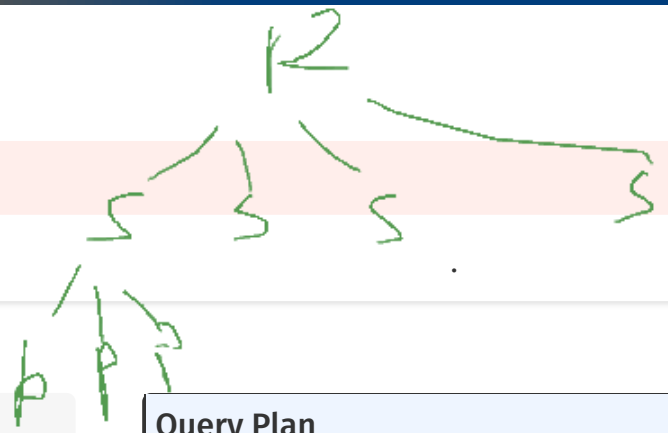
MultiColumn Index

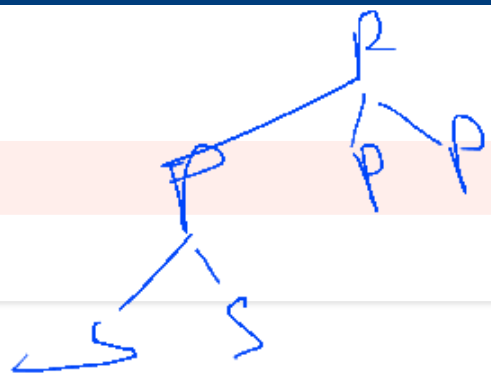
A multicolumn index can be used for index scan ***even with partial condition***.

Prefix Index

Question

Find the quantity of items with id 7





SQL

```
EXPLAIN SELECT s.s_qty
FROM stocks s
WHERE                    s.i_id = '7';
```

Query Plan
Seq Scan on stocks s (cost=0.00..804.40 rows=88 width=2)
Filter: (i_id = 7)

Caution

Only works if the condition involves the **prefix of the multicolumn index**.

stocks

Index-Only

Question

Find the **item id** of items on warehouse with id 123.

SQL

```
EXPLAIN SELECT s.i_id
FROM stocks s
WHERE s.w_id = '123'                    ;
```

Query Plan

```
Index Only Scan using stocks_pkey on stocks s
(cost=0.29..7.09 rows=160 width=4)
  Index Cond: (w_id = 123)
```

Index-Only Scan

Depending on the query, the scan can be done only within the index **without accessing the data**.

stocks_pkey

Index-Only

Question

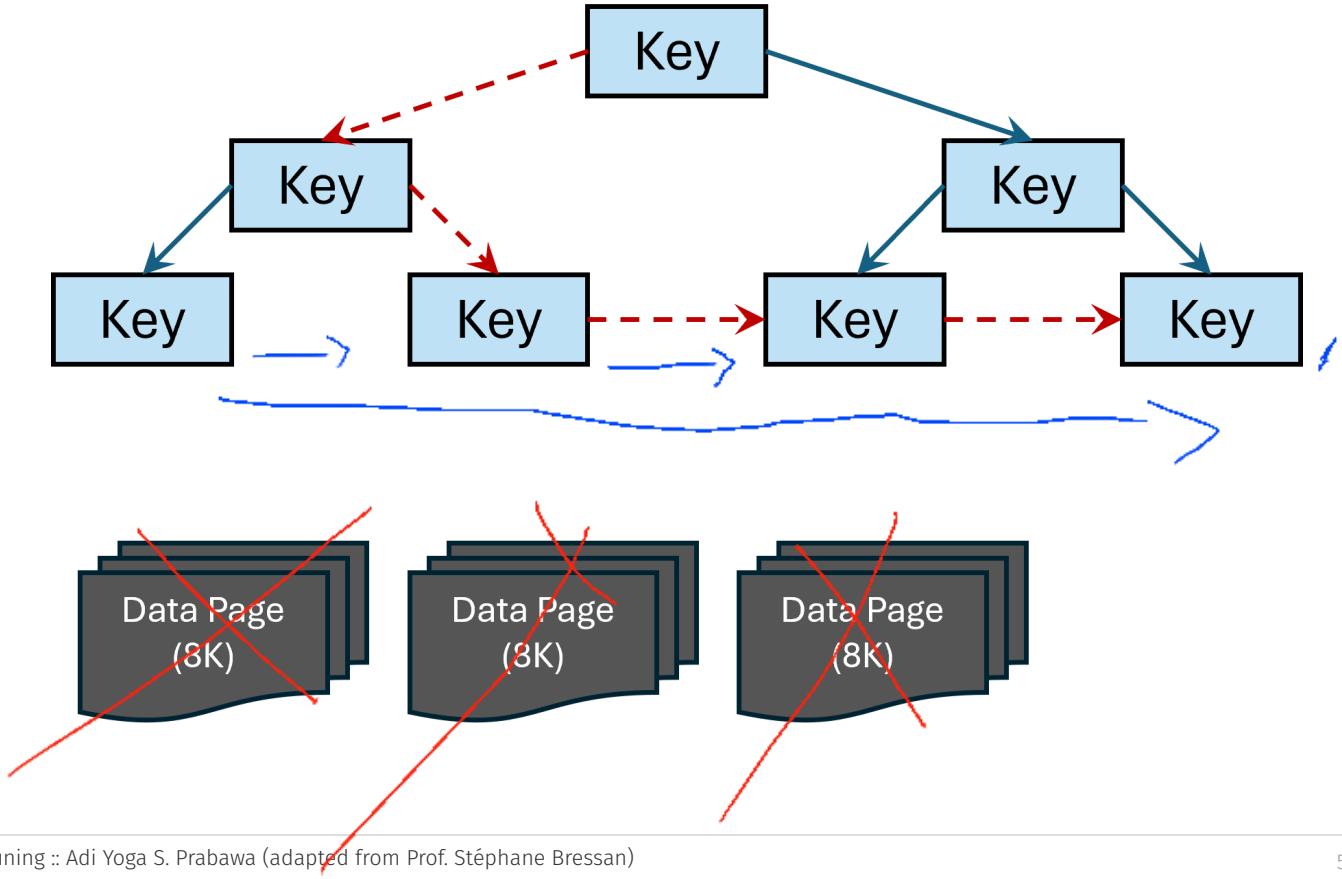
Find the **item id** of items on warehouse with id 123.

SQL

```
EXPLAIN SELECT s.i_id
FROM stocks s
WHERE s.w_id = '123'                    ;
```

Index-Only Scan

Depending on the query, the scan can be done only within the index **without accessing the data**.



Index-Only MultiColumn

Question

Find the warehouse id of items with id 7.

SQL

```
EXPLAIN SELECT s.w_id
FROM stocks s
WHERE                    s.i_id = '7';
```

Query Plan
Seq Scan on stocks s (cost=0.00..804.40 rows=88 width=4))
Filter: (i_id = 7)

Still Prefix

Again, for multicolumn index, it depends on the prefix.

stocks

More Index

Tuning to Index-Only

We can make the optimizer favor **index-only** scans by including more columns in the index.

Create Index

```
CREATE INDEX stocks_s_qty ON stocks(s_qty) INCLUDE (w_id, i_id);
```

SQL

```
EXPLAIN SELECT *
FROM stocks s
WHERE s_qty = 999;
```

Seq Scan
(no index)

Query Plan
Index Only Scan using stocks_s_qty on stocks s (cost=0.29..5.01 rows=41 width=10)
Index Cond: (s_qty = 999)

stocks_s_qty

```
postgres=# exit
```

```
Press any key to continue . . .
```