CS4221
Relational Databases II. Turning Strategies

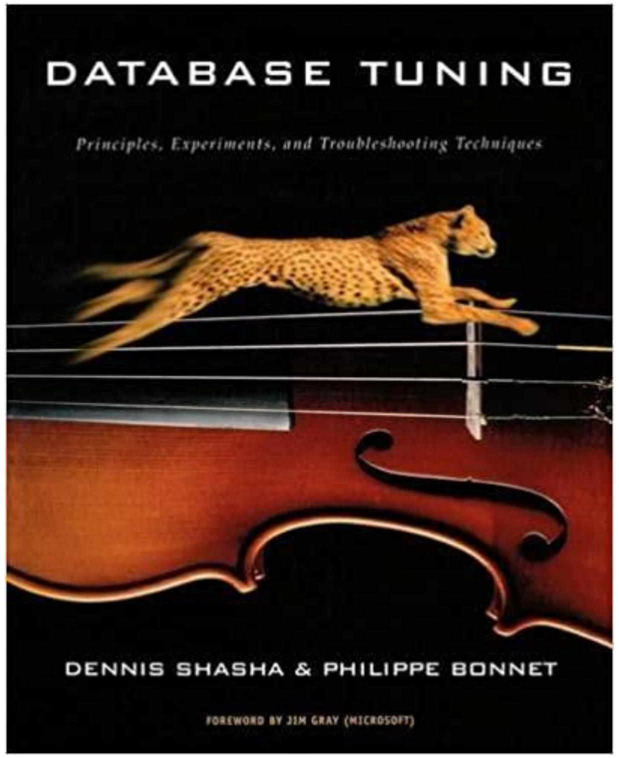
Yao LU

2024 Semester 2

National University of Singapore
School of Computing

“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 […]”

Database Tuning,
Dennis Shasha and Philippe Bonnet



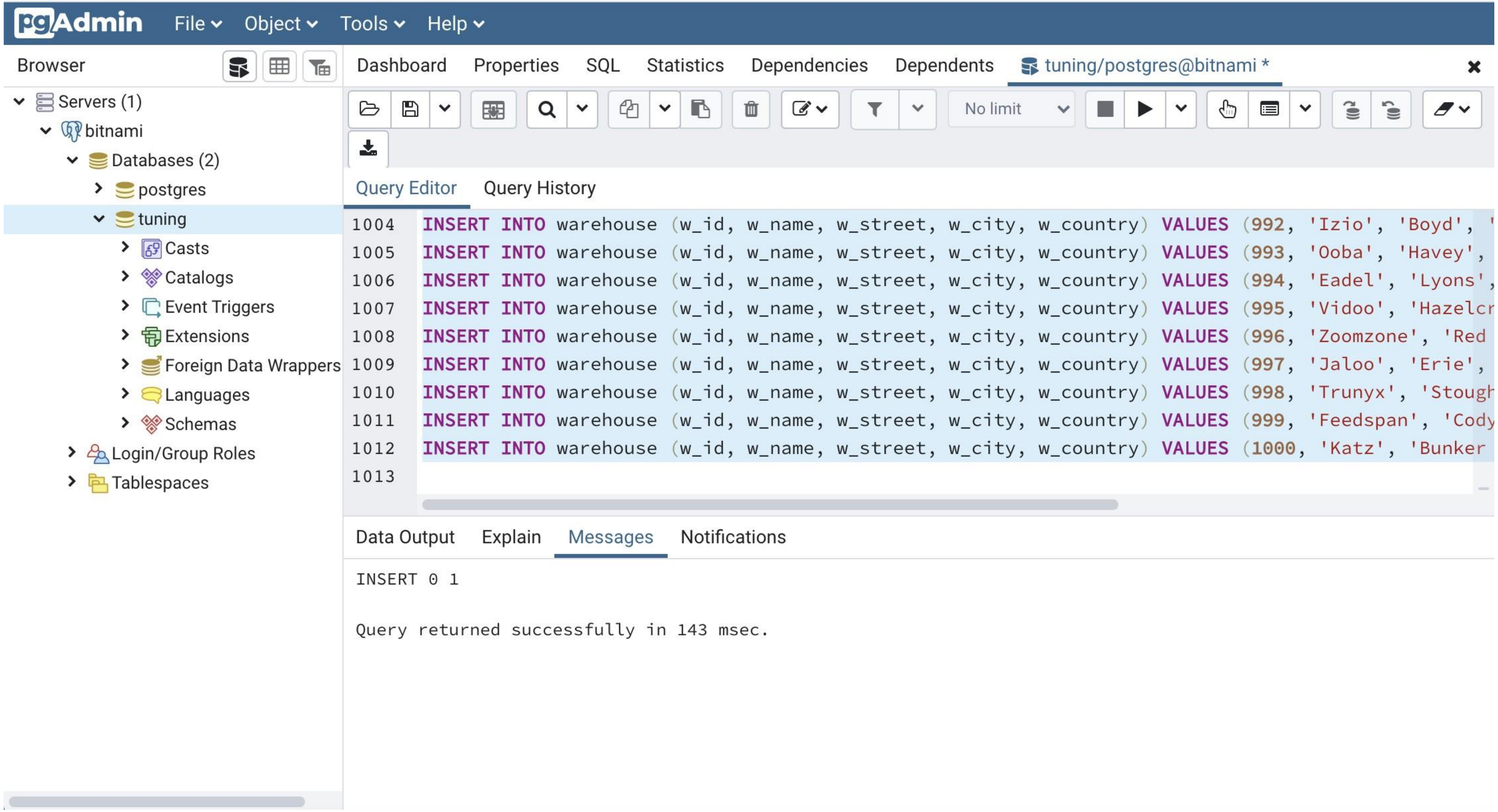
Schema

- Warehouses

Warehouses have a unique identifier, a name and a location defined by a street, city and country.

```sql
CREATE TABLE warehouse (
  w_id INTEGER PRIMARY KEY,
  w_name VARCHAR(50) NOT NULL,
  w_street VARCHAR(50) NOT NULL,
  w_city VARCHAR(50) NOT NULL,
  w_country CHAR(50) NOT NULL);
```

Schema


pgAdmin
File
Object
Tools
Help
Browser
Servers (1)
bitnami
Databases (2)
postgres
tuning
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Schemas
Login/Group Roles
Tablespaces
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
tuning/postgres@bitnami *
No limit
Query Editor
Query History
1004 INSERT INTO warehouse (w_id, w_name, w_street, w_city, w_country) VALUES (992, 'Izio', 'Boyd',
1005 INSERT INTO warehouse (w_id, w_name, w_street, w_city, w_country) VALUES (993, 'Ooba', 'Havey',
1006 INSERT INTO warehouse (w_id, w_name, w_street, w_city, w_country) VALUES (994, 'Eadel', 'Lyons',
1007 INSERT INTO warehouse (w_id, w_name, w_street, w_city, w_country) VALUES (995, 'Vidoo', 'Hazelcr
1008 INSERT INTO warehouse (w_id, w_name, w_street, w_city, w_country) VALUES (996, 'Zoomzone', 'Red
1009 INSERT INTO warehouse (w_id, w_name, w_street, w_city, w_country) VALUES (997, 'Jaloo', 'Erie',
1010 INSERT INTO warehouse (w_id, w_name, w_street, w_city, w_country) VALUES (998, 'Trunyx', 'Stough
1011 INSERT INTO warehouse (w_id, w_name, w_street, w_city, w_country) VALUES (999, 'Feedspan', 'Cody
1012 INSERT INTO warehouse (w_id, w_name, w_street, w_city, w_country) VALUES (1000, 'Katz', 'Bunker
1013
Data Output
Explain
Messages
Notifications
INSERT 0 1
Query returned successfully in 143 msec.

Schema

```sql
SELECT *
FROM warehouse;
```

warehouse				
w_id	w_name	w_street	w_city	w_country
301	’Schmedeman’	’Sunbrook’	’Singapore’	’Singapore’
1	’DabZ’	’Green’	’Patemon’	’Indonesia’
43	’Agimba’	’Heath’	’Cikaludan’	’Indonesia’
... 1005 rows				

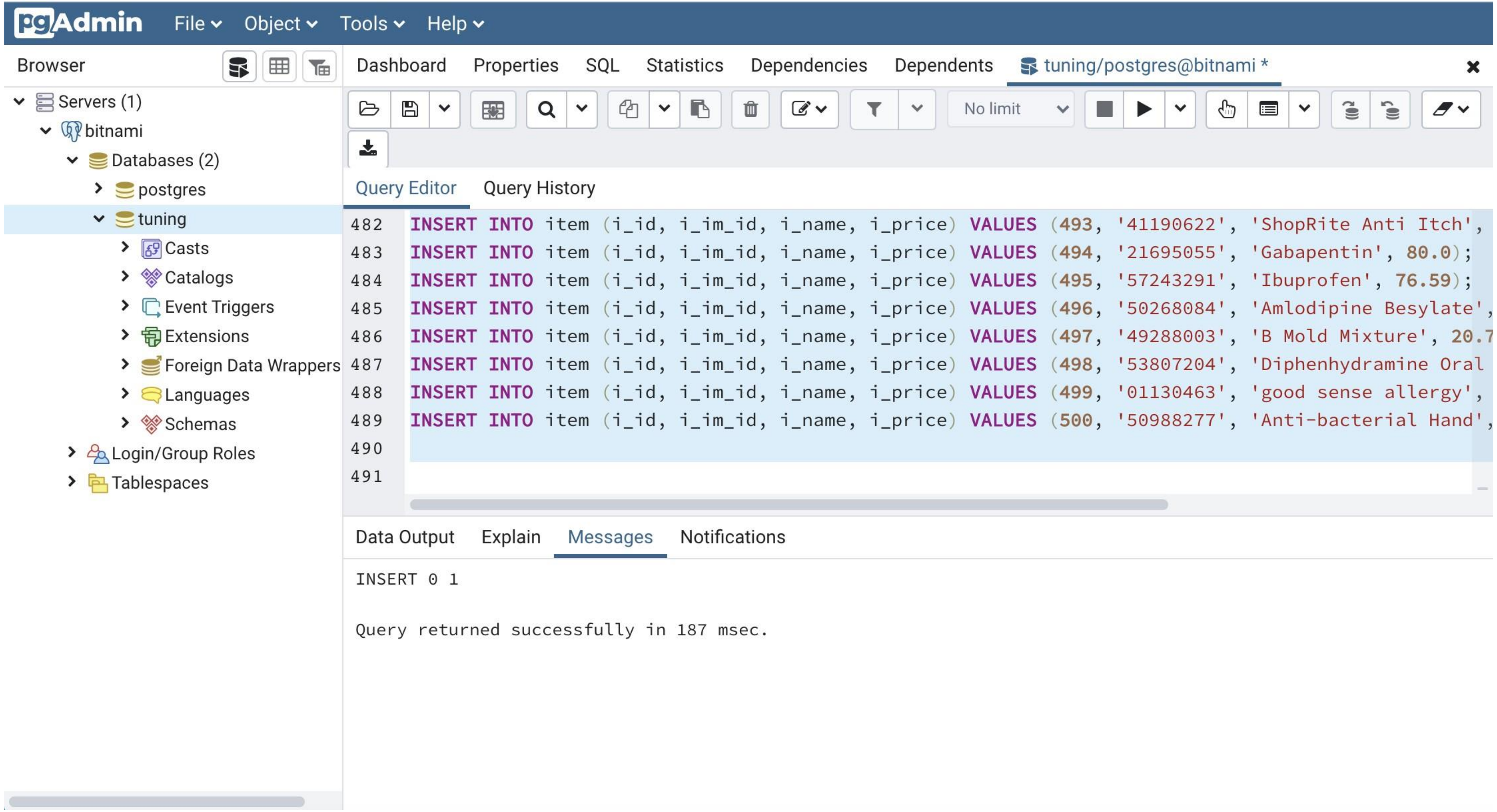
Schema

- Items

Items have a unique identifier, a unique image identifier, a name and a price.

```sql
CREATE TABLE item (
i_id INTEGER PRIMARY KEY,
i_im_id VARCHAR(8) UNIQUE NOT NULL,
i_name VARCHAR(50)  NOT NULL,
i_price NUMERIC(5, 2)  NOT NULL CHECK(i_price > 0));
```

Schema



pgAdmin
File
Object
Tools
Help
Browser
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
tuning/postgres@bitnami *
No limit
Servers (1)
bitnami
Databases (2)
postgres
tuning
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Schemas
Login/Group Roles
Tablespaces
Query Editor
Query History
482 INSERT INTO item (i_id, i_im_id, i_name, i_price) VALUES (493, '41190622', 'ShopRite Anti Itch',
483 INSERT INTO item (i_id, i_im_id, i_name, i_price) VALUES (494, '21695055', 'Gabapentin', 80.0);
484 INSERT INTO item (i_id, i_im_id, i_name, i_price) VALUES (495, '57243291', 'Ibuprofen', 76.59);
485 INSERT INTO item (i_id, i_im_id, i_name, i_price) VALUES (496, '50268084', 'Amlodipine Besylate',
486 INSERT INTO item (i_id, i_im_id, i_name, i_price) VALUES (497, '49288003', 'B Mold Mixture', 20.7
487 INSERT INTO item (i_id, i_im_id, i_name, i_price) VALUES (498, '53807204', 'Diphenhydramine Oral
488 INSERT INTO item (i_id, i_im_id, i_name, i_price) VALUES (499, '01130463', 'good sense allergy',
489 INSERT INTO item (i_id, i_im_id, i_name, i_price) VALUES (500, '50988277', 'Anti-bacterial Hand',
490
491
Data Output
Explain
Messages
Notifications
INSERT 0 1
Query returned successfully in 187 msec.

Schema

```sql
SELECT *
FROM item;
```

item			
i_id	i_im_id	i_name	i_price
1	’35356226’	’Indapamide’	95.23
6	’11822073’	’miconazole 1’	73.35
10	’60429082’	’Glipizide’	12.62
... 483 rows			

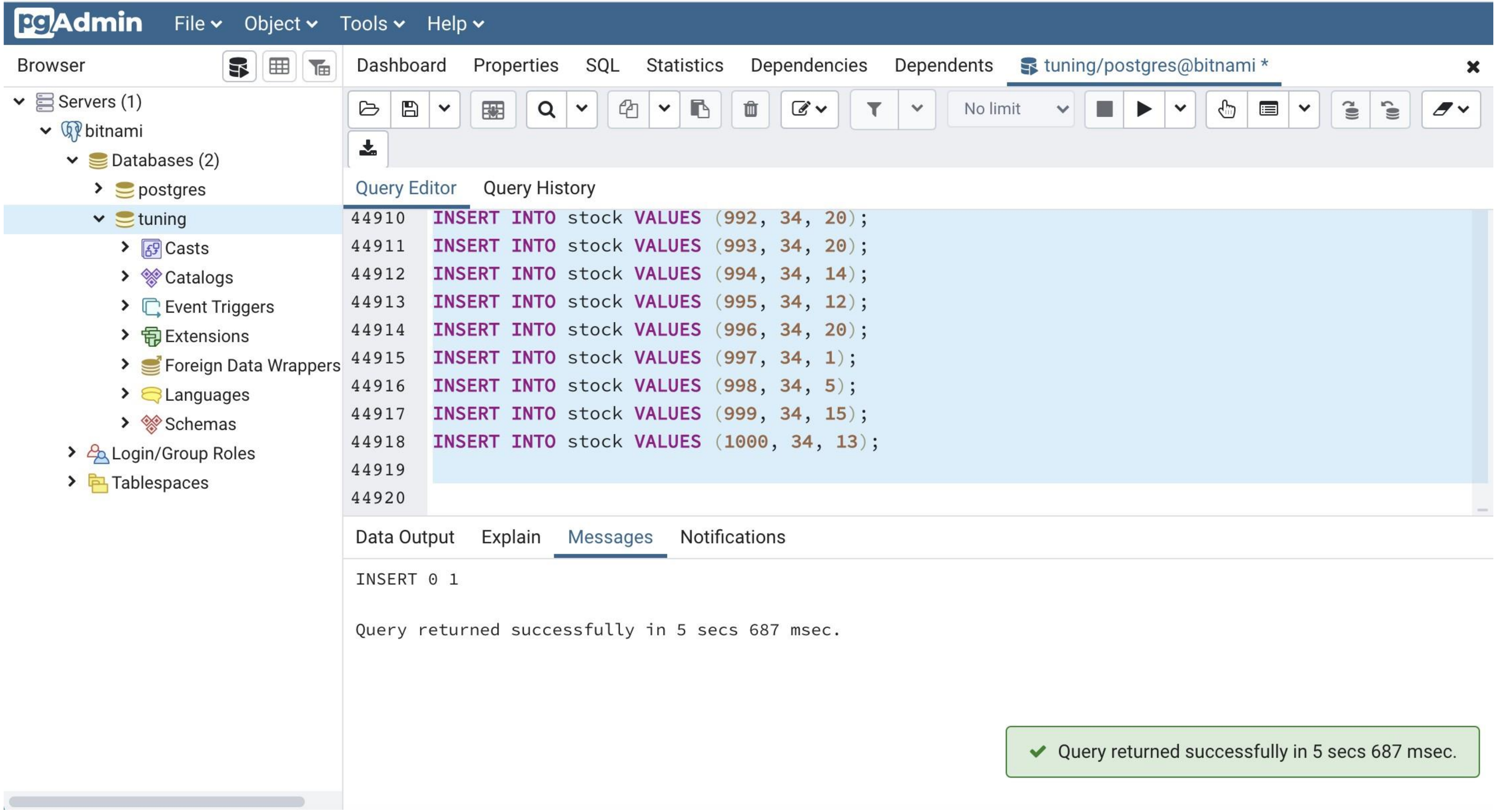
Schema

- Stocks

For each item available we record the quantity in stock in each warehouse. If an item is not available in a warehouse, then there is no entry for this pair. The quantity is always equal to or greater than 1.

```
CREATE TABLE stock (
w_id INTEGER REFERENCES warehouse(w_id),
i_id INTEGER REFERENCES item(i_id),
s_qty SMALLINT NOT NULL CHECK(s_qty > 0),
PRIMARY KEY (w_id, i_id));
```

Schema


pgAdmin
File
Object
Tools
Help
Browser
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
tuning/postgres@bitnami *
Servers (1)
bitnami
Databases (2)
postgres
tuning
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Schemas
Login/Group Roles
Tablespaces
No limit
Query Editor
Query History
44910 INSERT INTO stock VALUES (992, 34, 20);
44911 INSERT INTO stock VALUES (993, 34, 20);
44912 INSERT INTO stock VALUES (994, 34, 14);
44913 INSERT INTO stock VALUES (995, 34, 12);
44914 INSERT INTO stock VALUES (996, 34, 20);
44915 INSERT INTO stock VALUES (997, 34, 1);
44916 INSERT INTO stock VALUES (998, 34, 5);
44917 INSERT INTO stock VALUES (999, 34, 15);
44918 INSERT INTO stock VALUES (1000, 34, 13);
44919
44920
Data Output
Explain
Messages
Notifications
INSERT 0 1
Query returned successfully in 5 secs 687 msec.
Query returned successfully in 5 secs 687 msec.

Schema

```sql
SELECT *
FROM stock;
```

stock		
w_id	i_id	s_qty
301	5	760
301	4	938
243	352	515
... 44912 rows		

Inside PostgreSQL

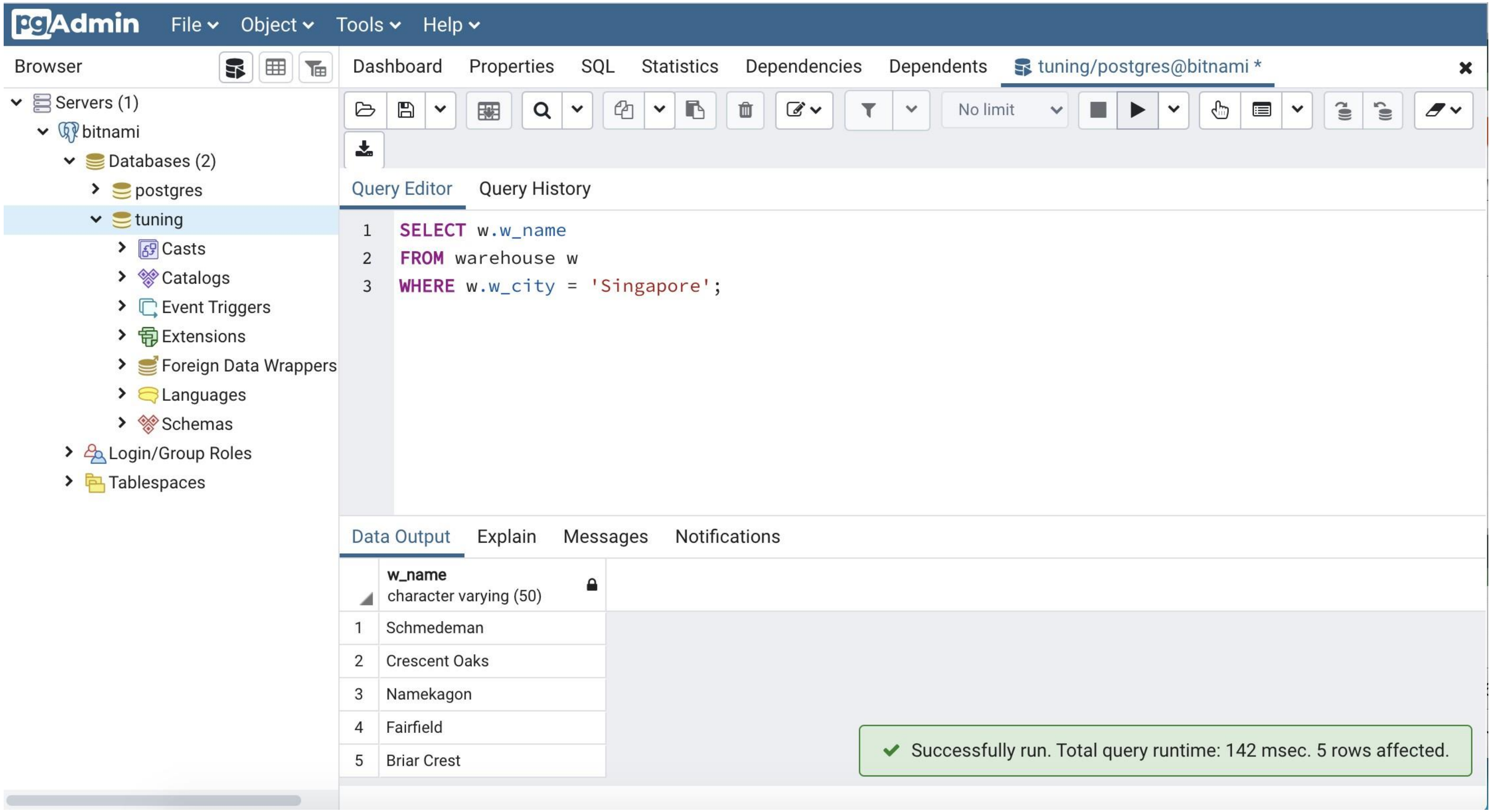
- What Happens to a Query?

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

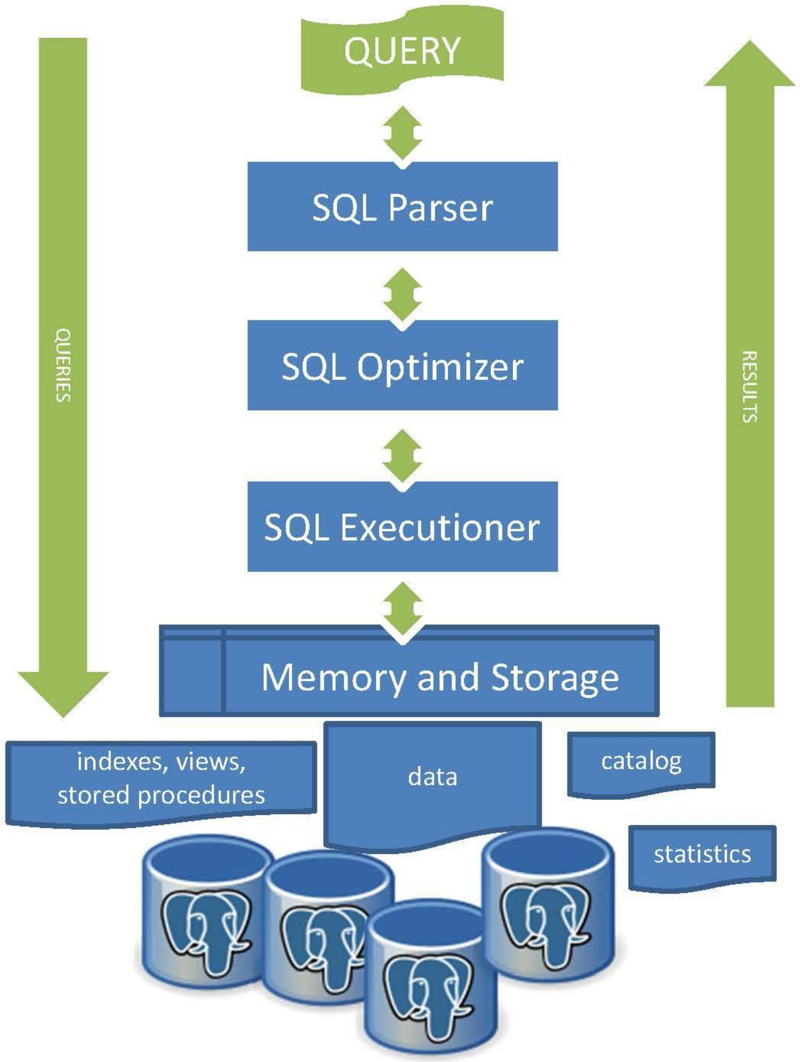
```
SELECT w.w_name
FROM warehouse w
WHERE w.w_city = 'Singapore';
```

w_name
character varying(50)
"Schmedeman"
"Crescent Oaks"
"Namekagon"
"Fairfield"
"Briar Crest"

Inside PostgreSQL



Inside PostgreSQL



Inside PostgreSQL

- Query Planner/Optimizer

PostgreSQL query planner/optimizer tries and creates an optimal execution plan. An execution plan is a tree of physical algebraïc operators such as sequential scans, index scans, sorting and aggregation operators, nested loop, hash, and merge joins.
PostgreSQL query planner/optimizer uses the catalogue and statistics to estimate the cost of the possible plans and to find a plan with an estimated least cost.

- Query Executioner

PostgreSQL query executioner executes the execution plan. It accesses the data, indexes and stored functions.

- Timings

The total query runtime includes the planning time, the execution time, and the time spent communicating with the client.

Explain

- EXPLAIN displays the execution plan that the PostgreSQL query planner/optimizer generates for the supplied statement.

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

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

Explain

EXPLAIN displays the execution plan that PostgreSQL query planner/optimizer generates for the supplied statement. At each node of the execution plan, i.e. for each operator, it gives several estimates.

- Estimated start-up cost in units of disk page fetches by the node.
- Estimated total cost in units of disk page fetches by the node.

  Estimated number of rows output by the node.

  Estimated average width in bytes of rows output by the node

Explain

- The cost is estimated in units of disk page fetches.
- The cost is proportional to the time spent.
- The start-up cost (time expended before the output scan can start, e.g., time to do the sorting in a sort node)
- The total cost of a node includes the total cost of all its children.
- The total cost is an estimate. A query with a LIMIT clause, for example, may not pay the total cost.
- CPU effort is also estimated. It is converted into disk-page units using some fairly arbitrary fudge factors.
- The total cost of the root node does not include the transmission of results to the client.

Explain

- System Catalogs and Statistics

PostgreSQL query planner/optimizer uses statistics build (and maintained) by PostgreSQL.

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

For instance, the view pg_stats records that Singapore is a most common value of the column w_city with frequency of 0.00497512 (in a table of 1005 rows.) It also records the average width of columns.

See also other system catalogs and views such as pg_tables, pg_attribute, and pg_statistic.

ANALYZE

```sql
EXPLAIN ANALYZE SELECT w.w_name
FROM warehouse w
WHERE w.w_city = 'Singapore';
```

Query Plan
Seq Scan on warehouse w (cost=0.00..21.56 rows=5 width=7) (actual time=0.037..0.759 rows=5 loops=1)
Filter: ((w_city)::text = 'Singapore'::text) Rows Removed by Filter: 1000
Planning time: 0.122 ms
Execution time: 0.798 ms

ANALYZE

- EXPLAIN ANALYZE

EXPLAIN ANALYZE gives for each node estimates obtained by random sampling as well as actual numbers for start up and total cost, number of rows and number of executions.

  - Actual start-up time in milliseconds.
  - Actual total time in milliseconds.
  - Actual number of rows output by this plan node.
  - Actual number of executions of the node (for instance if an indexed scan is repeated).

EXPLAIN ANALYZE gives the planning and execution times.

ANALYZE

Planning time: 0.122 ms
Execution time: 0.798 ms

- EXPLAIN ANALYZE

EXPLAIN ANALYZE also gives the actual total planning and execution times in milliseconds. The total execution time includes execution start-up and shut-down time, as well as time spent processing the result rows.

ANALYZE

- Actual Performance

In order to get a good idea of performance, one should run the queries many times and look at an average. Statistics are gathered. Pages are brought to the main memory buffer. VACUUM reorganizes the data on a regular basis. The costs, the times, and the plan change accordingly.

We do not do that in these slides.

pgAdmin 4

- Explain Analyze

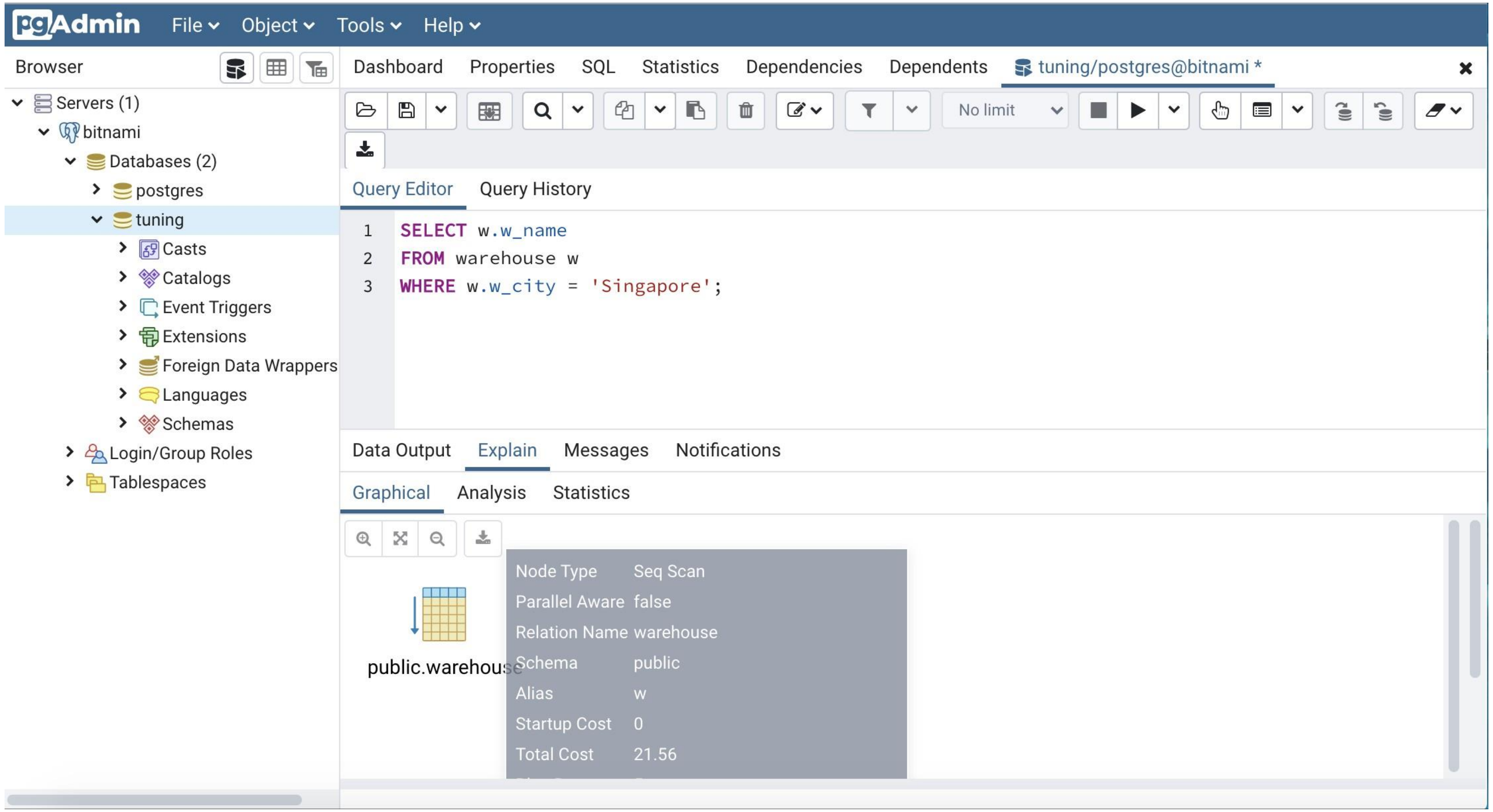
The Explain and Explain Analyze buttons in the toolbar of pgAdmin 4 generate the execution plan and the execution plan with execution timing, respectively. One can toggle the options to display in a verbose mode information about costs, buffers, and timings. The execution plan is represented in JSON.


```
"[{"Plan": {
"Node Type": "Seq Scan",
"Parallel Aware": false,
"Relation Name": "warehouse",
"Schema": "public",
"Alias": "w",
"Startup Cost": 0,
"Total Cost": 21.56,
"Plan Rows": 5,
"Plan Width": 7,
"Output": [
"w_name"
],
"Filter": "((w.w_city)::text = 'Singapore'::text)"}}]"
```

pgAdmin 4

- Graphical

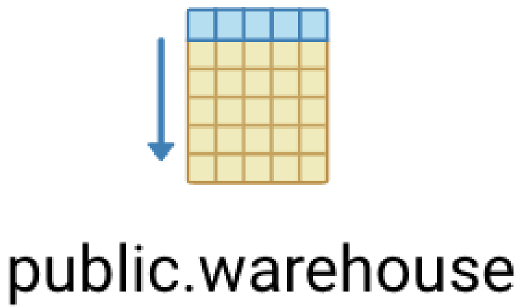
The Explain > Graphical tab shows a graphical version of the execution plan.



pgAdmin 4

- Graphical

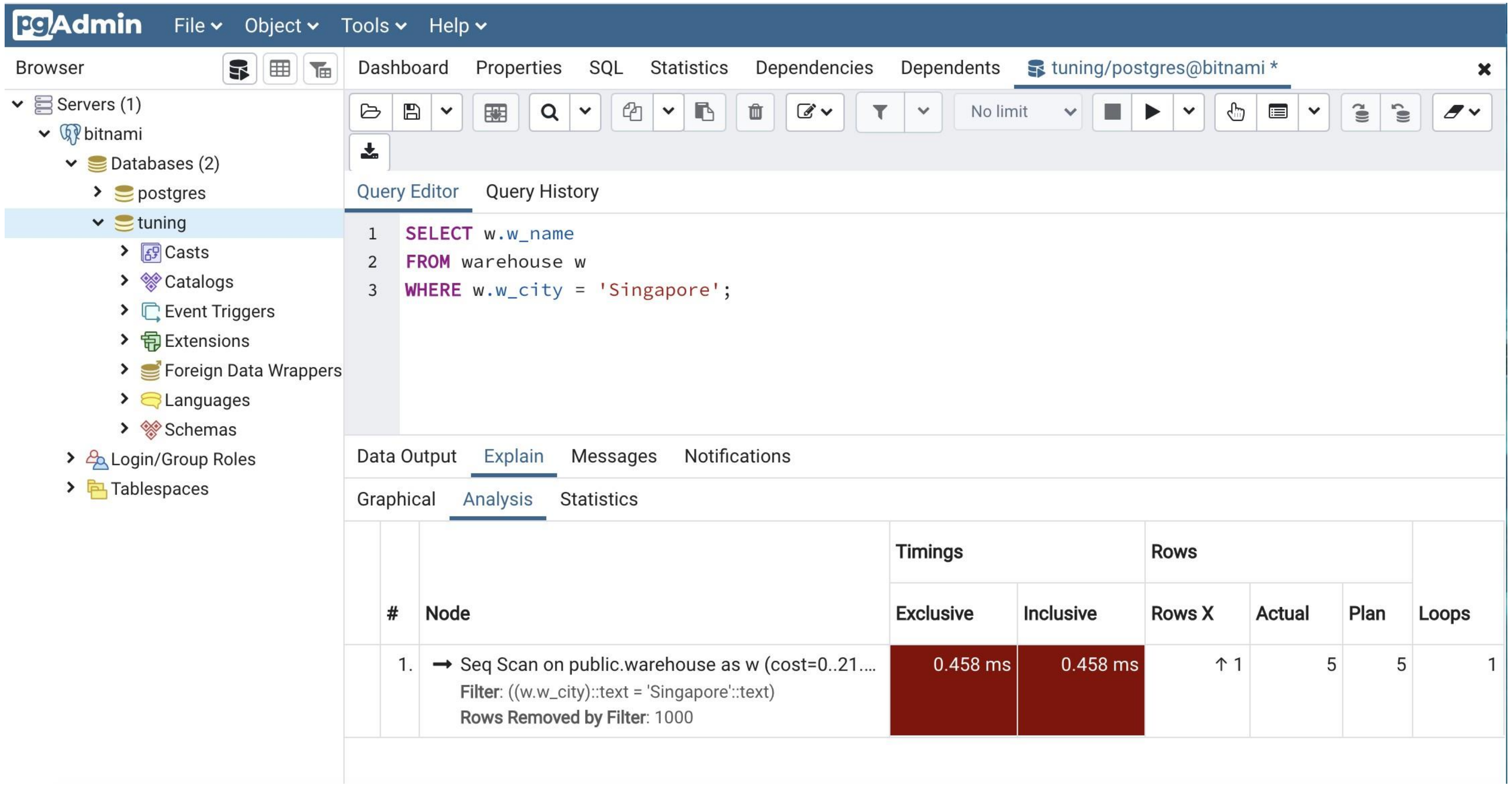
The `Explain > Graphical >` Download tab downloads a scalable vector graphics image of the graphical version of the execution plan.



pgAdmin 4

- Analysis

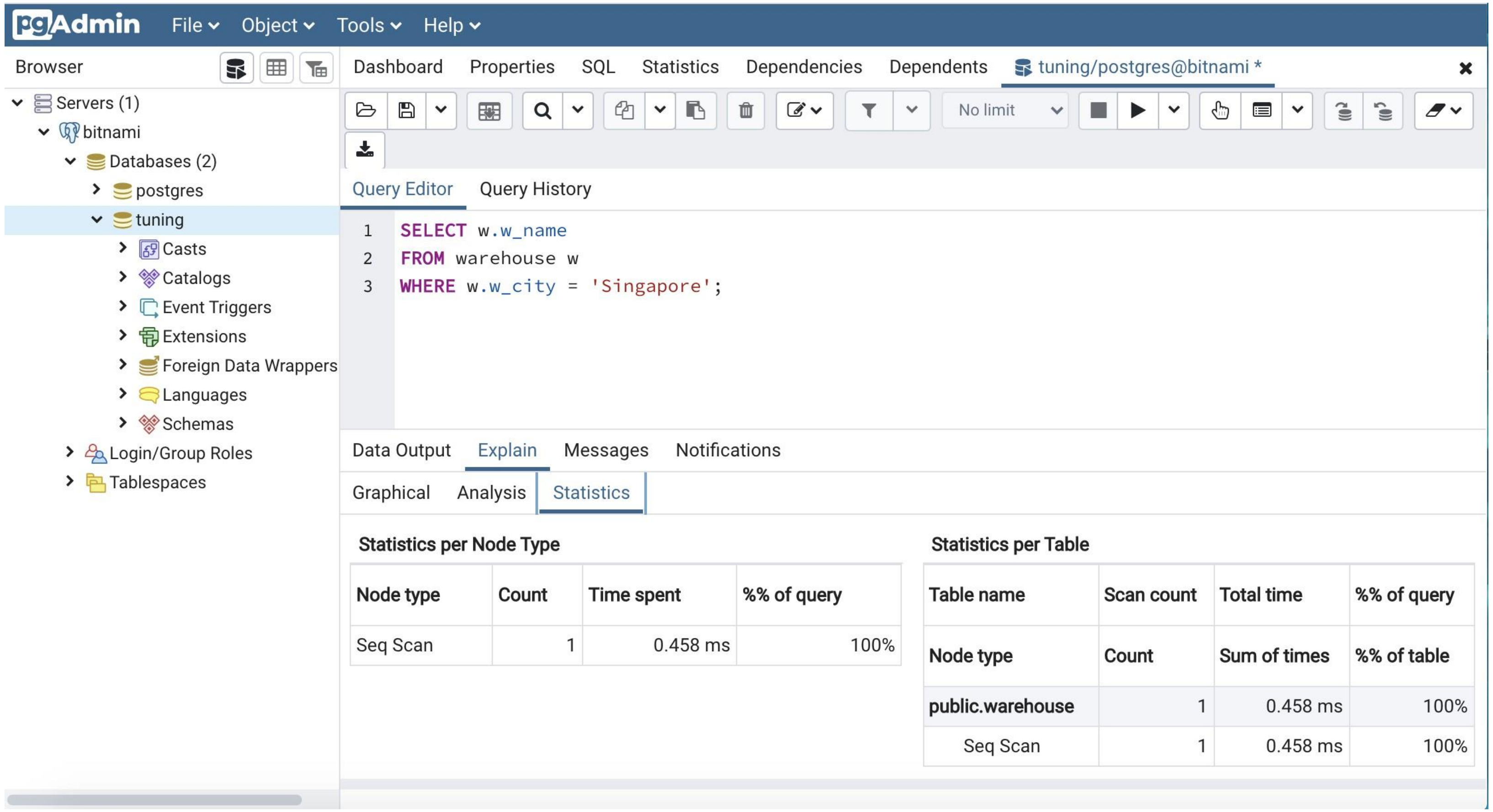
The Explain > Analysis tab shows the details of the execution plan in table format, with timings in Analyze mode. It is inspired by the online plan analysis tool “depesz” (see www.depesz.com and explain.depesz.com).



pgAdmin 4

- Statistics

The Explain > Statistics tab shows further statistics in Analyze mode .



Sequential Scan

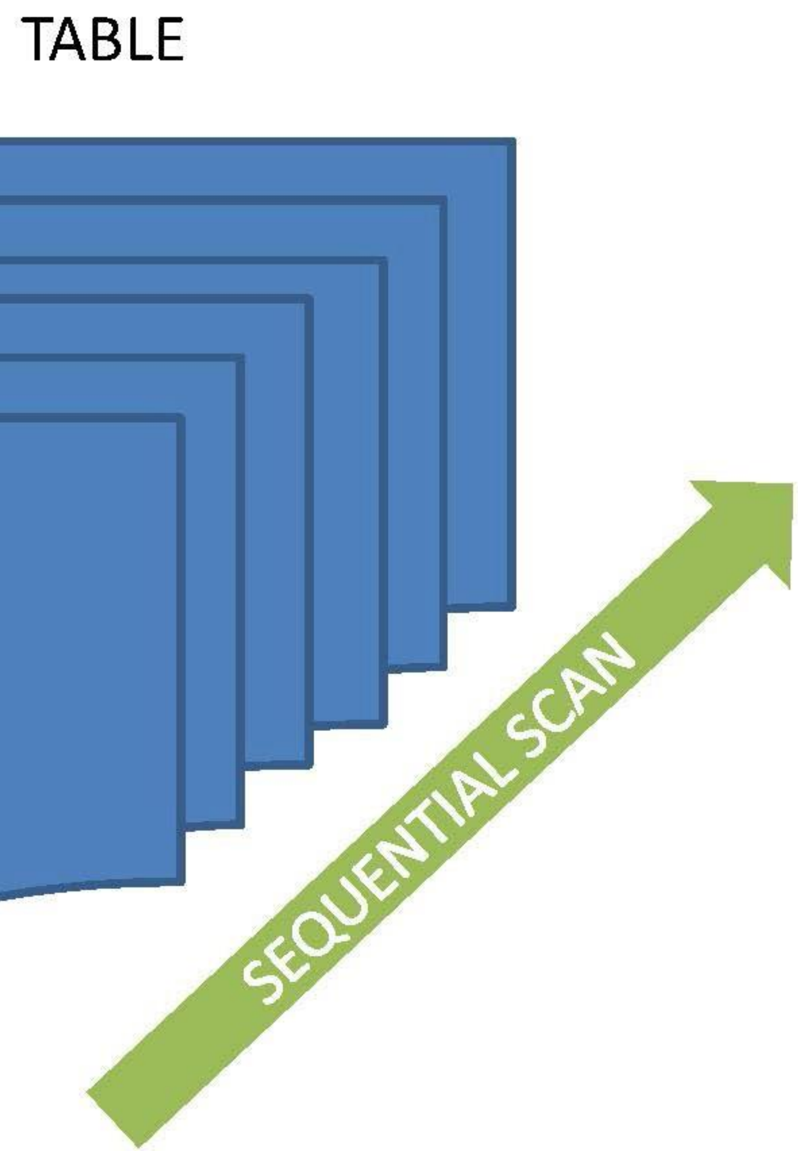
- Query

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

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

w_name
character varying(50)
"Schmedeman"
"Crescent Oaks"
"Namekagon"
"Fairfield"
"Briar Crest"

Sequential Scan



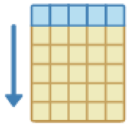
Sequential Scan

If the statistics indicate that the percentage of data to retrieve is large (more than 5% or so!) and it is scattered, it is not possible or worth trying to prepare and use another method than a sequential scan, then the optimizer uses a sequential scan.

Sequential Scan

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

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



public.warehouse

Sequential Scan

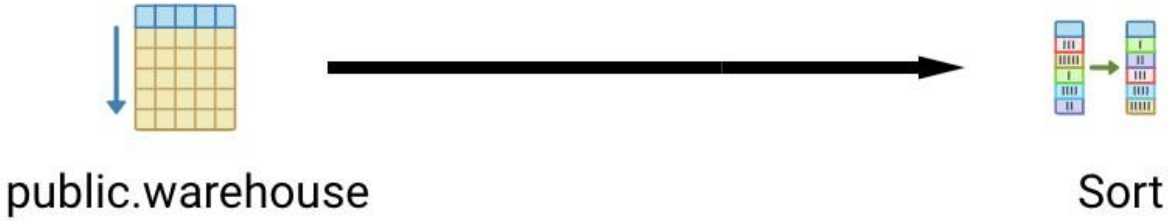
```sql
EXPLAIN ANALYZE SELECT w.w_name
FROM warehouse w
WHERE w.w_city = 'Singapore';
```

Query Plan
Seq Scan on warehouse w (cost=0.00..21.56 rows=5 width=7) (actual time=0.017..0.355 rows=5 loops=1)
Filter: ((w_city)::text = 'Singapore'::text)
Rows Removed by Filter: 1000
Planning time: 0.096 ms
Execution time: 0.372 ms

Sorting

```sql
EXPLAIN SELECT w.w_name
FROM warehouse 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 warehouse w (cost=0.00..21.56 rows=5 width=7)
Filter: ((w_city)::text = 'Singapore'::text)



Sorting

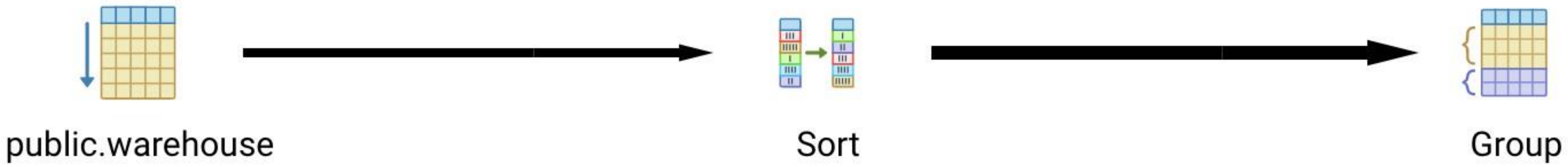
```sql
EXPLAIN ANALYZE SELECT w.w_name
FROM warehouse w
WHERE w.w_city = 'Singapore'
ORDER BY w.w_name;
```

Query Plan
Sort (cost=21.62..21.63 rows=5 width=7) (actual time=0.356..0.356 rows=5 loops=1)
Sort Key: w_name
Sort Method: quicksort Memory: 25kB
-> Seq Scan on warehouse w (cost=0.00..21.56 rows=5 width=7) (actual time=0.026..0.341 rows=5 loops=1)
Filter: ((w_city)::text = 'Singapore'::text)
Rows Removed by Filter: 1000
Planning time: 0.136 ms
Execution time: 0.377 ms

Sorting

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

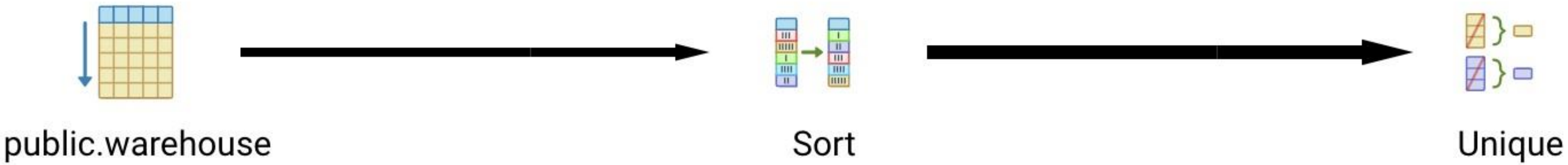
Query Plan
Unique (cost=21.62..21.65 rows=5 width=7)
-> Sort (cost=21.62..21.63 rows=5 width=7)
Sort Key: w_name
-> Seq Scan on warehouse w (cost=0.00..21.56 rows=5 width=7)
Filter: ((w_city)::text = 'Singapore'::text)



Sorting

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

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 warehouse w (cost=0.00..21.56 rows=5 width=7)
Filter: ((w_city)::text = 'Singapore'::text)



Index

- An index is a data structure that guides the 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 does not offer integrated index (data is the index) and only cluster indexes (data is organized according to the index) on demand and statically.

Index

- Primary Key

PostgreSQL automatically creates an index for each unique and primary key constraint. The index is used to enforce uniqueness (at extra cost for insertions and updates).

Index

- Foreign Key

PostgreSQL does not create an index for foreign key constraints.

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. It may be a good idea to create an index on the referencing columns. However, foreign key attributes are generally components of a composite key and are therefore indexed with a multicolumn index.

Index

- Finding the Existing Indexes

We create a view to gather information about the indexes from system tables.

```
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') column_names
FROM pg_index i, pg_class t,  pg_class ix
WHERE t.oid = i.indrelid AND ix.oid = i.indexrelid;
```

Index

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

table_name	index_name	is_unique	is_primary	column_names
"warehouse"	"warehouse_pkey"	t	t	"w_id"

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

table_name	index_name	is_unique	is_primary	column_names
"item"	"item_pkey"	t	t	"i_id"
"item"	"item_i_im_id_key"	t	f	"i_im_id"

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

table_name	index_name	is_unique	is_primary	column_names
"stock"	"stock_pkey"	t	t	"w_id, i_id"

Index

- Creating an Index

We can create an index on the i_price attribute of item.

```
CREATE INDEX i_i_price ON item(i_price);
```

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

table_name	index_name	is_unique	is_primary	column_names
"item"	"i_i_price"	f	f	"i_price"
"item"	"item_pkey"	t	t	"i_id"
"item"	"item_i_im_id_key"	t	f	"i_im_id"

Index

- Creating an Index: General Syntax

We highlight some improtant parameters of the CREATE INDEX command in PostgreSQL.

```
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 and other index types.
- predicate defines a partial index.

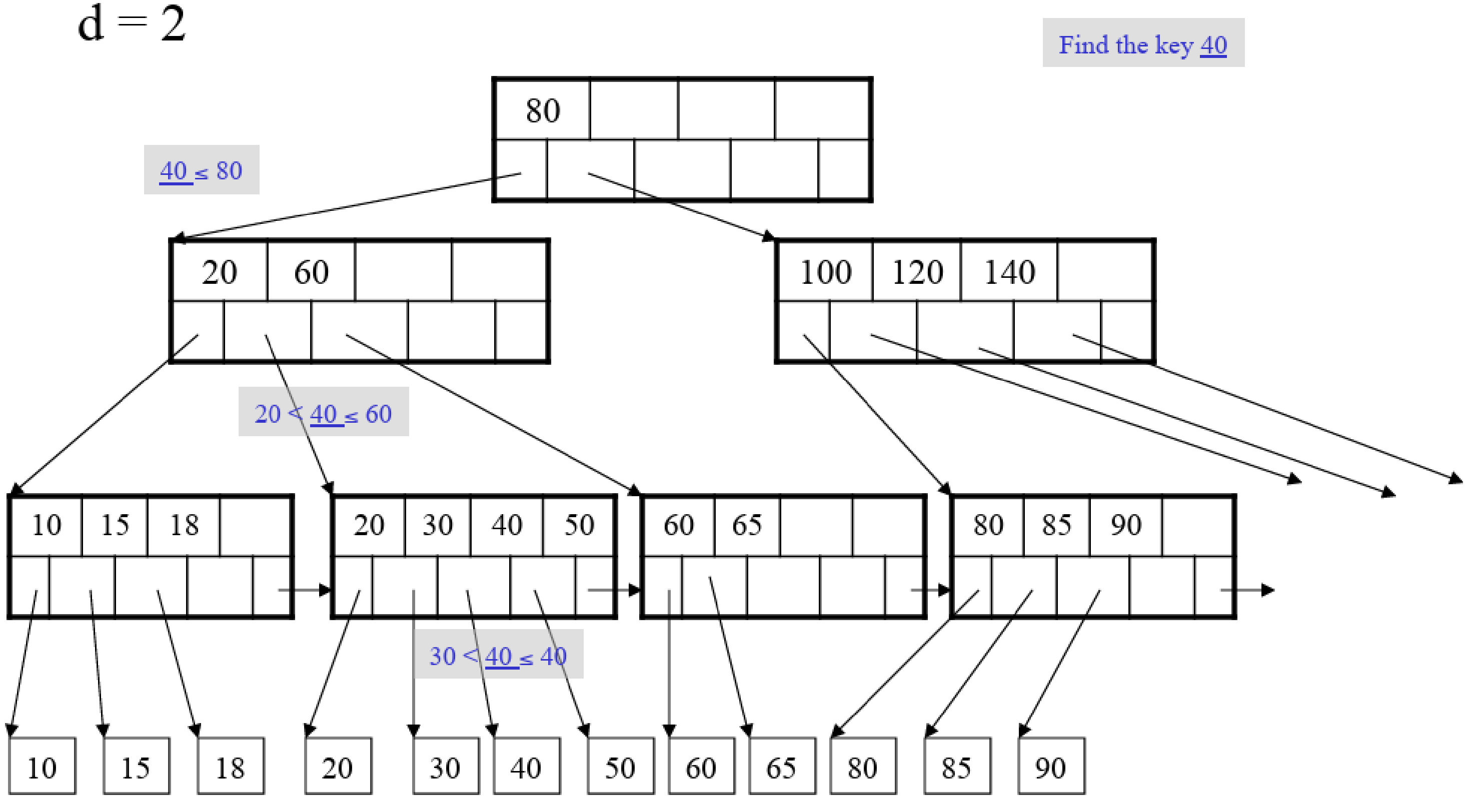
Index

- B+Trees

  What is a B+Tree index?

B+ Tree Example

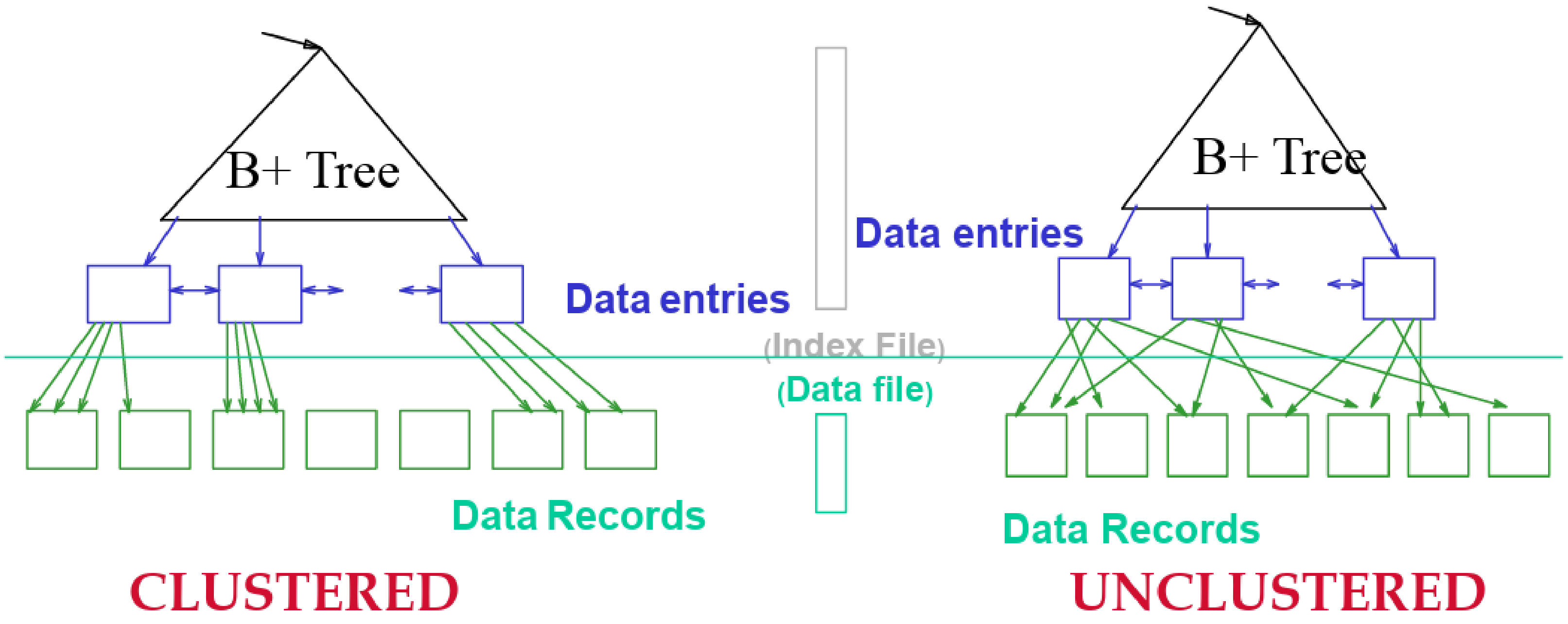
d = 2



Index

- Sparse vs Dense
  - What is the difference between a sparse and a dense index?
    Are PostgreSQL indexes sparse or dense?
- Clustered vs Unclustered
  - What is the difference between a clustered and an unclustered index?
    Are PostgreSQL indexes clustered or unclustered (see CLUSTER)?
- Primary vs Secondary
  - What is the difference between a primary and a secondary index?
  - Are PostgreSQL indexes primary or secondary?

Clustered vs. Unclustered Index



- More commonly, in a clustered B+ Tree index, **data entries are data records**

Index

- Covering
  - What is a covering index?
  - Can PostgreSQL indexes be covering (see also INCLUDE in PostgreSQL 11)?

Index Scan

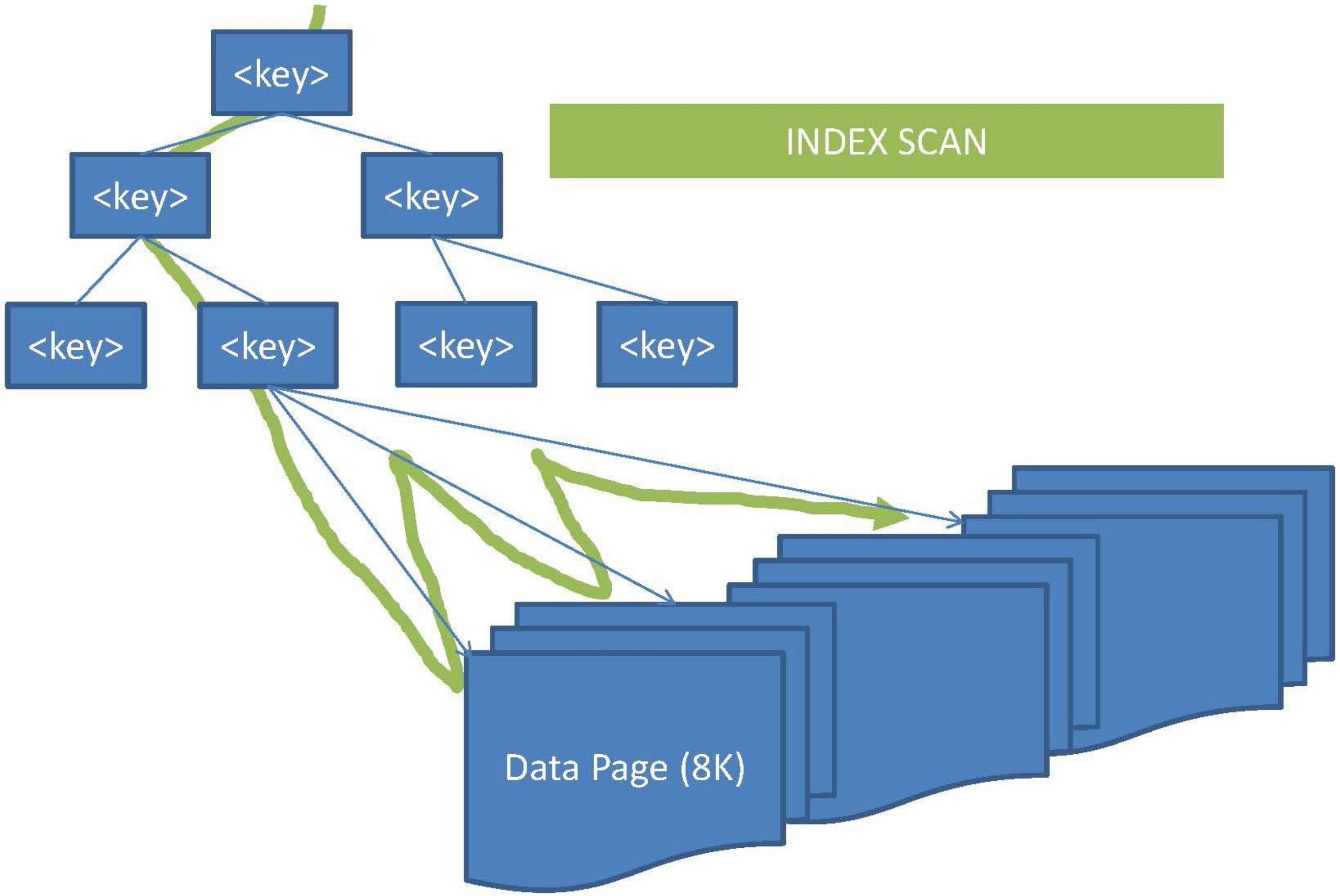
- Query

Find the name of the warehouse with identifier 123.

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

w_name character varying(50)
"Janyx"

Index Scan



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.

Index Scan

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

Query Plan
Index Scan using warehouse_pkey on warehouse w (cost=0.28..8.29 rows=1 width=7) (actual time=0.015..0.016 rows=1 loops=1)
Index Cond: (w_id = 123)
Planning time: 0.255 ms
Execution time: 0.058 ms

Bitmap Heap Scan

- Creating an Index

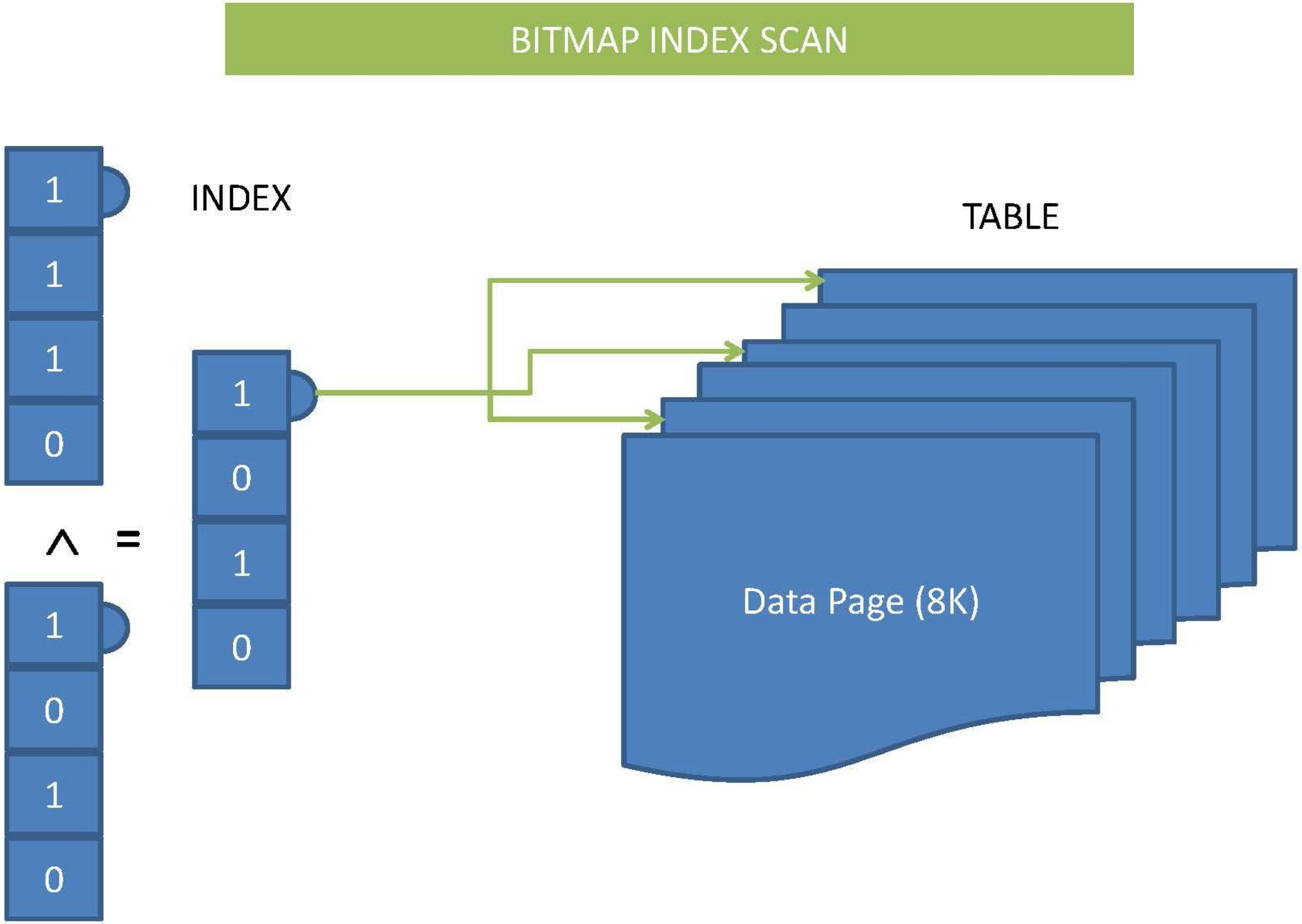
Create a B+Tree index (default) on the w_city attribute of warehouse.

```
CREATE INDEX i_w_city ON warehouse(w_city);
```

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

table_name	index_name	is_unique	is_primary	column_names
"warehouse"	"warehouse_pkey"	t	t	"w_id"
"warehouse"	"i_w_city"	f	f	"w_city"

Bitmap Heap Scan



Bitmap Heap Scan

- 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.

Bitmap Heap Scan

```sql
EXPLAIN ANALYZE SELECT w.w_name
FROM warehouse w
WHERE w.w_city = 'Singapore';
```

Query Plan
Bitmap Heap Scan on warehouse w (cost=4.31..12.38 rows=5 width=7) (actual time=0.055..0.057 rows=5 loops=1)
Recheck Cond: ((w_city)::text = 'Singapore'::text)
Heap Blocks: exact=1
-> Bitmap Index Scan on i_w_city (cost=0.00..4.31 rows=5 width=0) (actual time=0.046..0.046 rows=5 loops=1)
Index Cond: ((w_city)::text = 'Singapore'::text)
Planning time: 0.504 ms
Execution time: 0.092 ms

The Bitmap Index Scan is implemented by a Bitmap Index Scan followed by a Bitmap Heap Scan in PostgreSQL.

Bitmap Heap Scan

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

```
EXPLAIN ANALYSZE ELECT w.w_name
FROM warehouse w
WHERE w.w_city = 'Singapore';

SELECT * FROM warehouse;

CLUSTER warehouse USING i_w_city;

EXPLAIN ANALYZE SELECT w.w_name
FROM warehouse w
WHERE w.w_city = 'Singapore';

SELECT * FROM warehouse;
```