Indexing Explained

Robert Ivens

JBS Group, Partner

“An index makes your query fast”

An index can speed up...

- Searching
  - WHERE clause
- Joining tables
- Sorting data
  - ORDER BY
  - UNION

Servoy doesn't create indexes

Indexes are not in the SQL standard

Placing indexes is not exact science

(other than primary keys)

How does an index work?

RowID	Age
1	58
2	2
3	10
4	83
5	46
6	67

Table scan

No index applied

How does an index work?

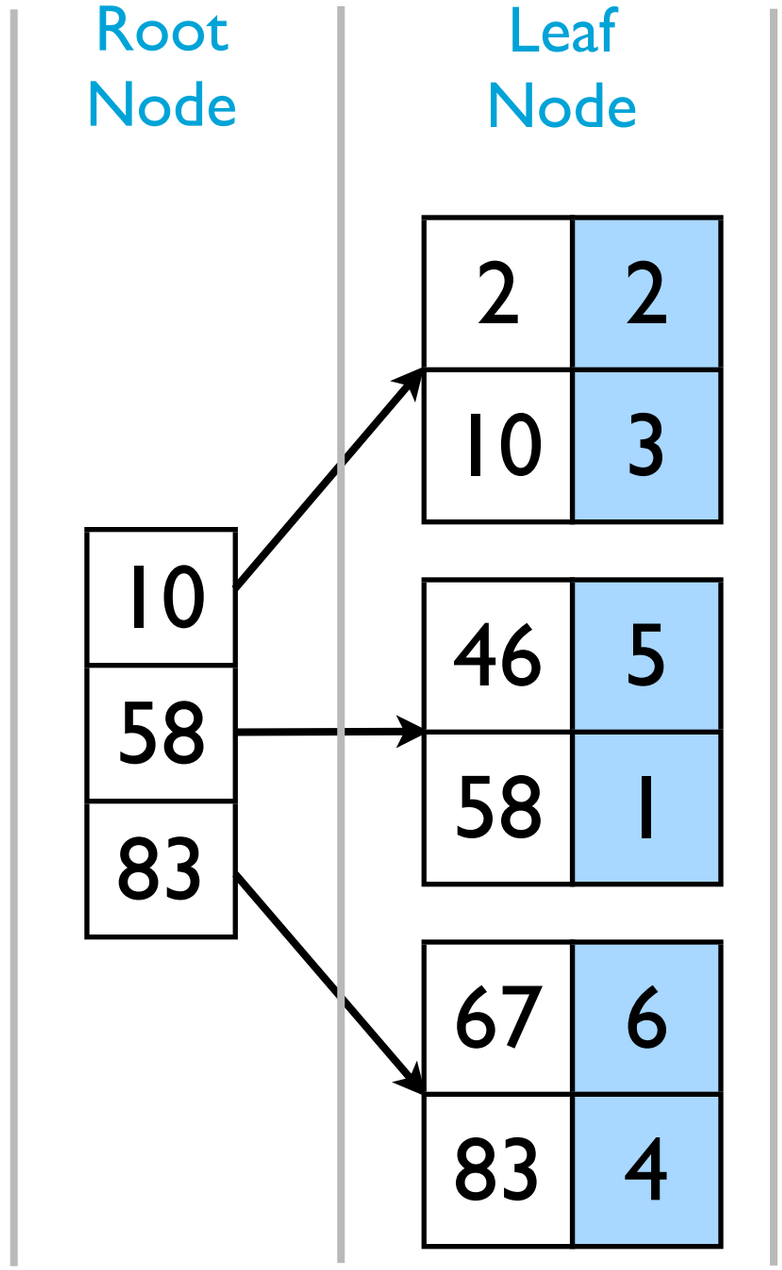
RowID	Age
1	58
2	2
3	10
4	83
5	46
6	67

```
create index <name> on
<table> (age)
```

Apply Index

How does an index work?

RowID	Age
1	58
2	2
3	10
4	83
5	46
6	67

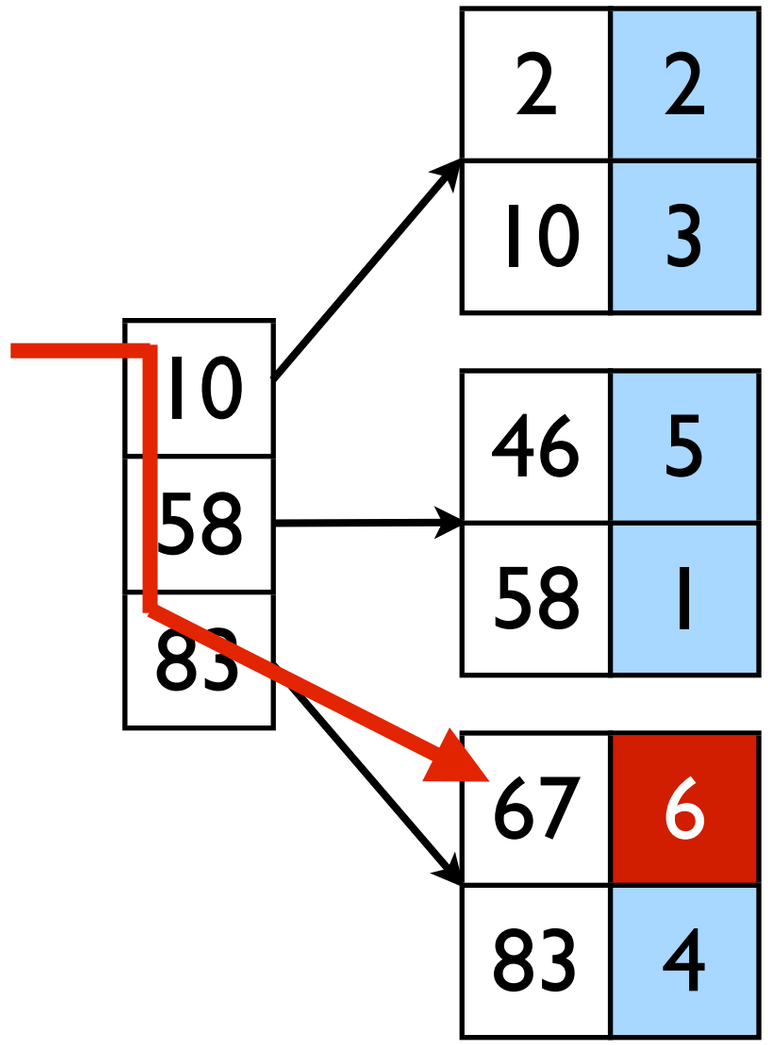


Balanced Tree Index (B-tree)

How does an index work?

Find row with age 67

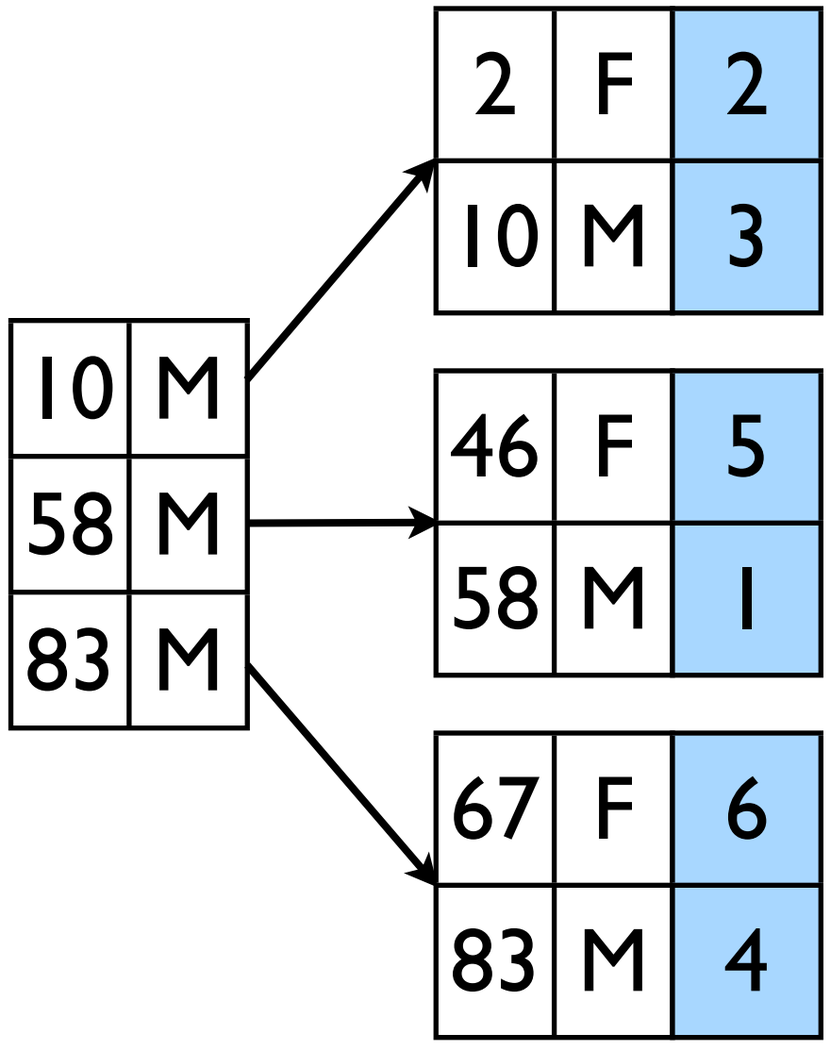
RowID	Age
1	58
2	2
3	10
4	83
5	46
6	67



B-tree traversal = FAST!

How does an index work?

RowID	Age	Sex
1	58	M
2	2	F
3	10	M
4	83	M
5	46	F
6	67	F

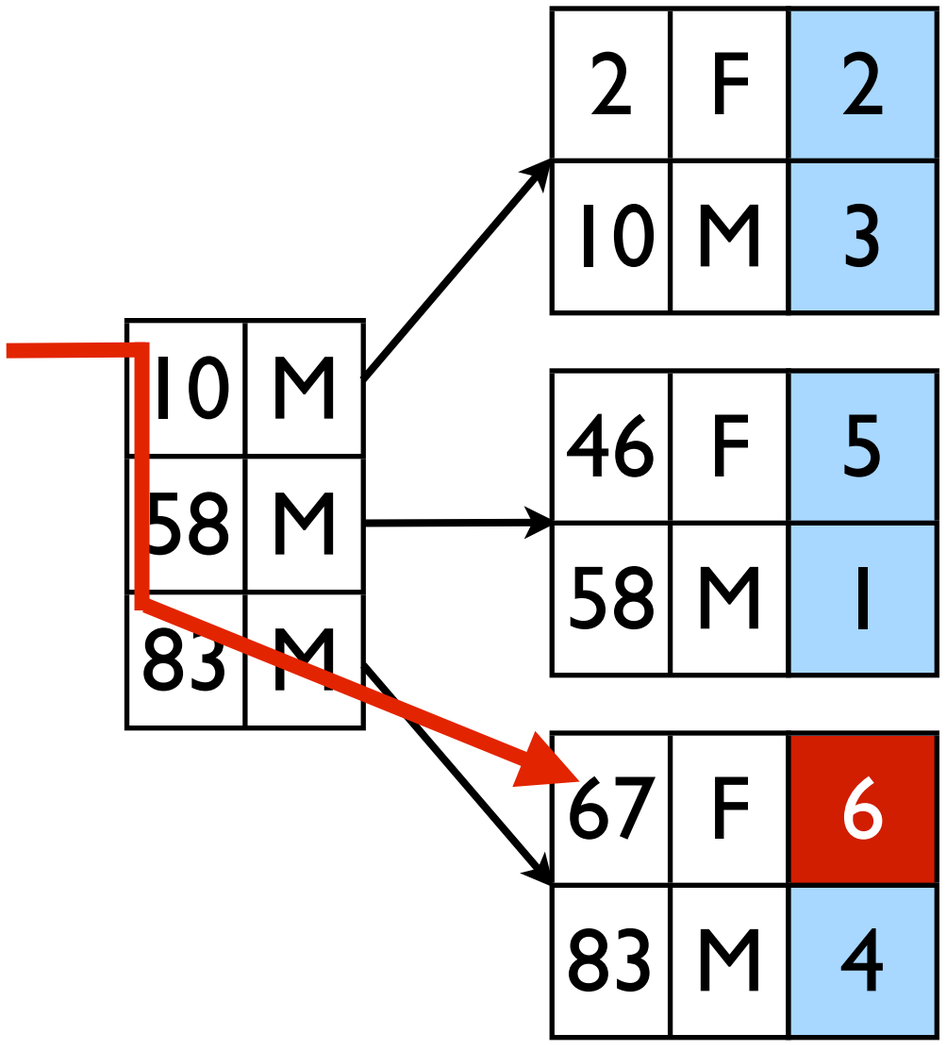


Multi-column index (Age, Sex)

How does an index work?

Find row with age 67 and Female

RowID	Age	Sex
1	58	M
2	2	F
3	10	M
4	83	M
5	46	F
6	67	F

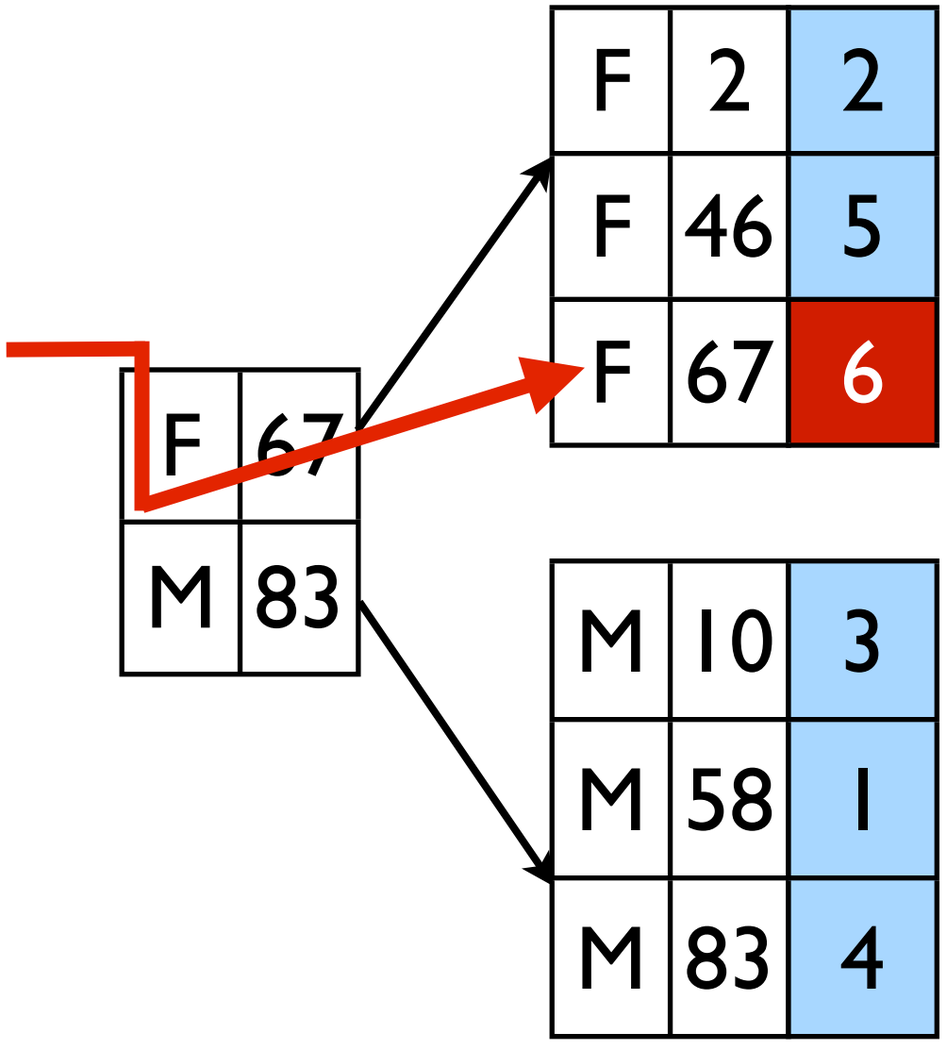


Multi-column index (Age, Sex)

How does an index work?

Find row with age 67 and Female

RowID	Age	Sex
1	58	M
2	2	F
3	10	M
4	83	M
5	46	F
6	67	F

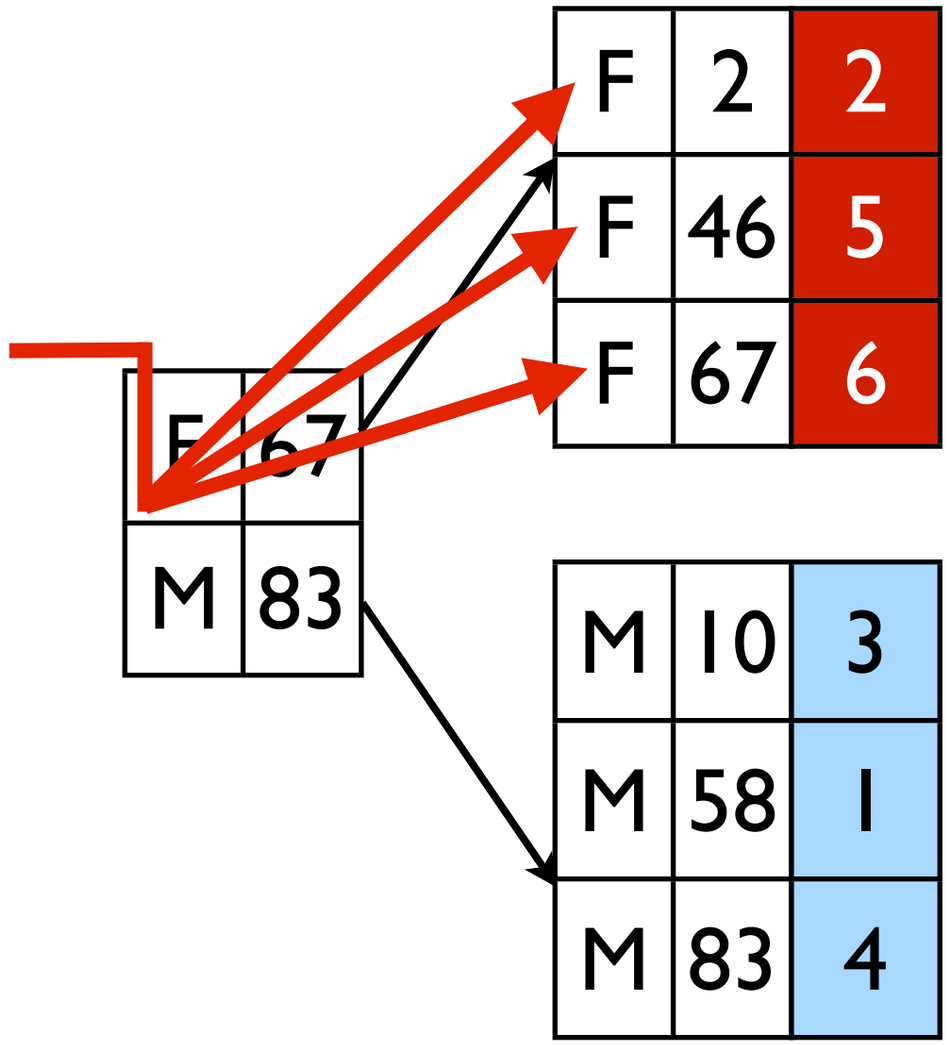


Multi-column index (Sex, Age)

How does an index work?

Find row with Female

RowID	Age	Sex
1	58	M
2	2	F
3	10	M
4	83	M
5	46	F
6	67	F

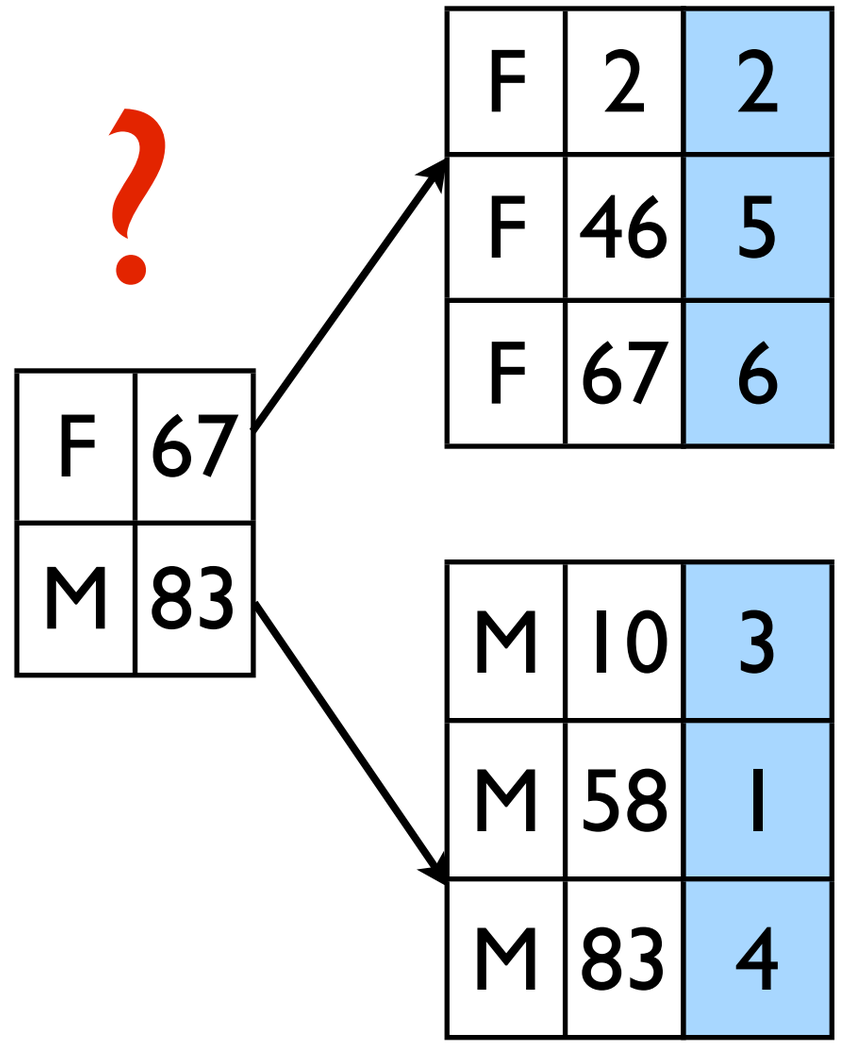


Multi-column index (Sex, Age)

How does an index work?

Find row with age 67

RowID	Age	Sex
1	58	M
2	2	F
3	10	M
4	83	M
5	46	F
6	67	F



Multi-column index (Sex, Age)

How does an index work?

Find row with age 67

RowID	Age	Sex
1	58	M
2	2	F
3	10	M
4	83	M
5	46	F
6	67	F

Table scan

Multi-column index (Sex, Age)

How does an index work?

B-tree depth	Index entries
3	64
4	256
5	1024
6	4096
7	16.384
8	65.536
9	262.144
10	1.048.576

B-tree logarithmic scalability

How does an index work?

Find rows with age 67

RowID	Age
1	67
2	67
3	67
4	83
5	67
6	67

67	1,2,3,5,6
83	4

Low cardinality = SLOW

How does an index work?

Find rows with age 67

RowID	Age
1	67
2	67
3	67
4	83
5	67
6	67

Table scan = FAST

How does it know what to use?

Cost based query planning

- PostgreSQL keeps meta data about your data
  - how many rows in table
  - what is the most common value in a column
  - number of distinct values in a column
  - keeps statistics on the data by sampling (ANALYZE)
- cost of using an index vs table scan
- HDD vs SSD
- cost is abritary unit

Prepared Statement gotcha

- SELECT *

  FROM tableName

  WHERE myColumn=?

- Query planner doesn’t look at the passed values, only the SQL

- So, what is the cardinality of ‘?’

How to know why an index is used or not?

Index usage

- EXPLAIN <query>
  - shows the query plan and rows it expects
  - Doesn’t run the query itself
- EXPLAIN ANALYZE <query>
  - Shows the query plan and stats it expects
  - Runs the query and returns the actual stats
  - Doesn’t return the query data

Index usage

- pg_stat_* views (PostgreSQL)
  - pg_stat_user_tables
    - idx_scan
    - seq_scan
    - last_autoanalyze
    - last_analyze
  - pg_stat_user_indexes
    - indexrelname
    - idx_scan

Explain Analyze

```
                                                   QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=146.63..148.65 rows=808 width=138) (actual time=55.009..55.012 rows=71 loops=1)
   Sort Key: n.nspname, p.proname, (pg_get_function_arguments(p.oid))
   Sort Method: quicksort  Memory: 43kB
   ->  Hash Join  (cost=1.14..107.61 rows=808 width=138) (actual time=42.495..54.854 rows=71 loops=1)
         Hash Cond: (p.pronamespace = n.oid)
         ->  Seq Scan on pg_proc p  (cost=0.00..89.30 rows=808 width=78) (actual time=0.052..53.465 rows=2402 loops=1)
               Filter: pg_function_is_visible(oid)
         ->  Hash  (cost=1.09..1.09 rows=4 width=68) (actual time=0.011..0.011 rows=4 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 1kB
               ->  Seq Scan on pg_namespace n  (cost=0.00..1.09 rows=4 width=68) (actual time=0.005..0.007 rows=4 loops=1)
                     Filter: ((nspname <> 'pg_catalog'::name) AND (nspname <> 'information_schema'::name))
```

?????

Explain

```
explain select * from test where i = 1;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on test  (cost=0.00..40.00 rows=12 width=4)
   Filter: (i = 1)
(2 rows)
```

Sequential scan (Table scan)

Explain

```
explain select * from test where i = 1;
                        QUERY PLAN
-----------------------------------------------------------
-----------------------------------------------------------
 Index Scan using test_pkey on test  (cost=0.00..3.18 rows=12 width=4)
   Filter: (i = 1)
(2 rows)
```

Index scan

Explain Analyze

```
------------------------------------------------------------------------
Sort  (cost=146.63..148.65 rows=808 width=138) (actual
  Sort Key: n.nspname, p.proname, (pg_get_function_argu
  Sort Method: quicksort  Memory: 43kB
  ->  Hash Join  (cost=1.14..107.61 rows=808 width=138)
        Hash Cond: (p.pronamespace = n.oid)
        ->  Seq Scan on pg_proc p  (cost=0.00..89.30 ro
              Filter: pg_function_is_visible(oid)
        ->  Hash  (cost=1.09..1.09 rows=4 width=68) (ac
              Buckets: 1024  Batches: 1  Memory Usage:
              ->  Seq Scan on pg_namespace n  (cost=0.0
                    Filter: ((nspname <> 'pg_catalog'::
```

Node

Node

Node

Node

Explain Analyze

```
------------------------------------------------------------------------
Sort  (cost=146.63..148.65 rows=808 width=138) (actual
  Sort Key: n.nspname, p.proname, (pg_get_function_argu
  Sort Method: quicksort  Memory: 43kB
  ->  Hash Join  (cost=1.14..107.61 rows=808 width=138)
        Hash Cond: (p.pronamespace = n.oid)
        ->  Seq Scan on pg_proc p  (cost=0.00..89.30 ro
              Filter: pg_function_is_visible(oid)
        ->  Hash  (cost=1.09..1.09 rows=4 width=68) (ac
              Buckets: 1024  Batches: 1  Memory Usage:
              ->  Seq Scan on pg_namespace n  (cost=0.0
                    Filter: ((nspname <> 'pg_catalog'::
```

Read from inside out

Explain

http://www.depesz.com/tag/unexplainable/

Functional Indexes

- SELECT *

  FROM myTable

  WHERE upper(name) = ?

- CREATE INDEX <name> ON <table> (upper(name))

- Functions have to be immutable (always return the same value)

Partial Indexes

- CREATE INDEX <name> ON <table> (<column>)

  WHERE creationDate >= ‘2015-01-01’

Index types (PostgreSQL)

- B-tree
- GiST (Generalized Search Tree)
  - used for full-text searches
- GIN (Generalized Inverted Index)
  - Can handle columns with more than one key (like arrays)
- BRIN (Block Range Indexes) NEW! in version 9.5

Block-Range Indexes (BRIN)

- Tiny indexes designed for large tables
- Minimum/maximum values stored for a range of blocks
  - default 1MB, 128 8K pages
- Allows skipping large sections of the table that cannot contain matching values
- Ideally for naturally-ordered tables, e.g. insert-only tables that are chronologically ordered
- Index is 0,003% the size of the heap
- Indexes are inexpensive to update
- Index every column at little cost
- Slower lookups than btree

BRIN Index example

```
CREATE TABLE brin_example AS
SELECT generate_series(1,100000000) AS id;

CREATE INDEX brin_index ON brin_example USING brin(id);

CREATE INDEX btree_index ON brin_example(id);
```

```
SELECT relname, pg_size_pretty(pg_relation_size(oid))
FROM pg_class
WHERE relname LIKE 'brin_%' OR relname = 'btree_index'
ORDER BY relname;

 relname       | pg_size_pretty
---------------+----------------
 brin_example  | 3457 MB
 brin_index    | 104 kB
 btree_index   | 2142 MB
```

Overview

- Place indexes where you need really them (Queries)
- Developers drive this requirement
- Data distribution in a table affects the usage of an index
  - This can change over time so keep tracking your index use
- Not using indexes can sometimes be faster then using indexes
- Indexes do have a cost (insert/update/delete)
- No every index type is fit for purpose, see what is best for your requirement (but B-tree is mostly the right choice)

Overview (continued)

- You can use functions in your indexes
  - as long as they always return the same value (immutable)
- You can use indexes over parts of your data
- LIKE clause can use indexes when you use it as a type-ahead (LIKE ‘servo%’)

Questions

Further reading

http://use-the-index-luke.com
http://www.depesz.com/tag/unexplainable/