2ndQuadrant®
PostgreSQL

Power of Indexing

Pavan Deolasee

PostgreSQL
the world's most advanced open source database

About me

- Contributing to PostgreSQL for over a decade
- Primary developer, committer and maintainer of Postgres-XL project
- Part of the team organising PGConf India for a few years
- Working with 2ndQuadrant for 3 years. Previously worked with VERITAS/Symantec and EnterpriseDB

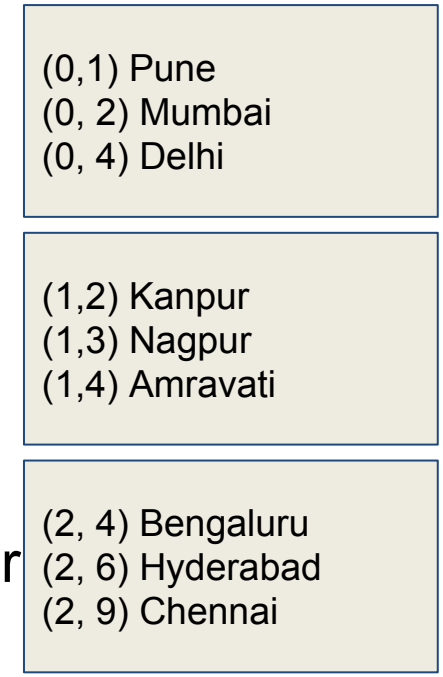
2ndQuadrant

- Major contributor to open source PostgreSQL
- Over 100k lines of code
- Logical Replication (10.0)
- Parallel Aggregates (9.6)
- Postgres-XL - MPP & horizontally scalable PostgreSQL (9.5)
- Multi Master Replication - BDR (9.4)
- Hot Standby (9.0)
- Point in time Recovery (8.0)

Accessing Data

Accessing Heap Data

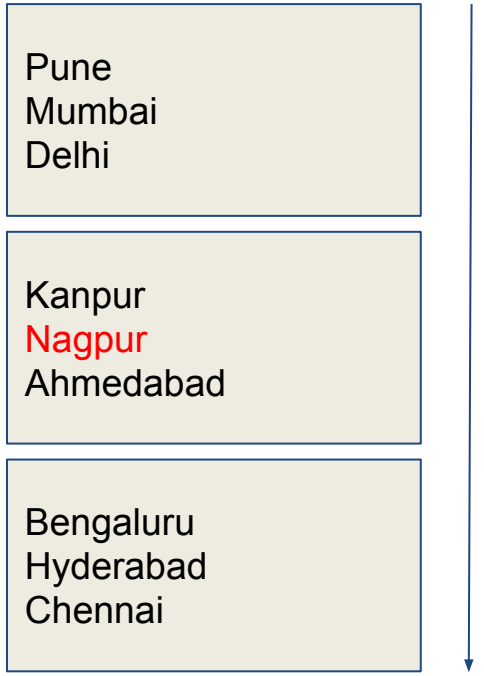
- Heap stores all tuples in one or more files
  - Each tuple has an address, what we call TID.
  - TID is the physical location of the tuple - `(blockNumber,offset)`
- Initially all tuples are stored in the order they were inserted
- Updates/delete, concurrent inserts can reorder tuples in non-deterministic way



Accessing Heap Data

- Sequential access to the data
- Must scan the entire table to find the required row(s)

```
SELECT * FROM emp WHERE
city = 'Nagpur';
```



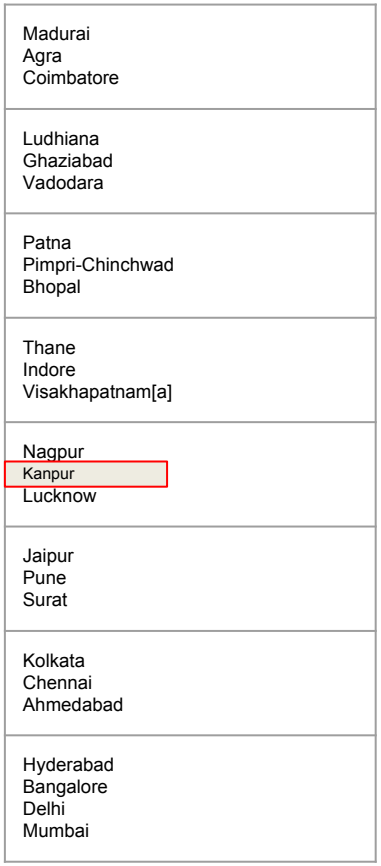
Sequential Scans

- Not always bad
- Sequential disk reads are much faster than random reads
- Works fine for small/narrow tables or when majority of the table data needs to be read

```
SELECT count(*) FROM emp;
```

Selection Queries

- What if you're looking for a single row in a very large table?
- Reading 1000s or millions of data blocks is not efficient
- Solution: build index



What is an index?

- A *secondary table* that stores and maintains key information, in a way so that it can be quickly searched.
- Also stores enough information to fetch the actual data from the main table
- Remember the “Index” printed at the end of the book? Book index has `(word, page);` we have `(key, pointer)`

Index Types

- B-Tree indexes
- Hash indexes
- BRIN indexes
- GIN and GiST indexes

B-Tree Indexes

B-Tree Index

- Tuples are sorted, ordered by key and stored in a secondary table
- Pointer to the heap tuple is stored along with the key. Pointer is nothing but the TID i.e. `(blockNumber, offset)` of the heap tuple.
- It's a tree - every branch has the same depth
- Default index type in PostgreSQL

Example

```
regression=# EXPLAIN SELECT count(*) FROM emp;
                          QUERY PLAN
-----------------------------------------------------------
 Aggregate  (cost=2.25..2.26 rows=1 width=8)
   ->  Seq Scan on emp  (cost=0.00..2.00 rows=100 width=0)
(2 rows)
regression=# EXPLAIN SELECT * FROM emp WHERE id = 2;
                       QUERY PLAN
-------------------------------------------------
 Seq Scan on emp  (cost=0.00..2.25 rows=1 width=45)
   Filter: (id = 2)
(2 rows)
```

Example

```
regression=# CREATE INDEX cityindex ON emp(city);
CREATE INDEX
regression=# CREATE UNIQUE INDEX pkindex ON emp(id);
CREATE INDEX
```

Example

```
regression=# EXPLAIN SELECT * FROM emp WHERE city = 'Nagpur';
                       QUERY PLAN
-------------------------------------------------------
 Seq Scan on emp  (cost=0.00..2.25 rows=4 width=45)
   Filter: (city = 'Nagpur'::text)
(2 rows)
regression=# EXPLAIN SELECT * FROM emp WHERE id = 2;
                       QUERY PLAN
-------------------------------------------------------
 Seq Scan on emp  (cost=0.00..2.25 rows=1 width=45)
   Filter: (id = 2)
(2 rows)
```

Example

```
regression=# ANALYZE emp;
ANALYZE
regression=# EXPLAIN SELECT * FROM emp WHERE id = 2;
                       QUERY PLAN
-------------------------------------------------------
 Seq Scan on emp  (cost=0.00..2.25 rows=1 width=45)
   Filter: (id = 2)
(2 rows)
regression=# EXPLAIN SELECT * FROM emp WHERE city = 'Nagpur';
                       QUERY PLAN
-------------------------------------------------------
 Seq Scan on emp  (cost=0.00..2.25 rows=4 width=45)
   Filter: (city = 'Nagpur'::text)
(2 rows)
```

Example

```
regression=# TRUNCATE emp;

TRUNCATE TABLE


-- Insert 10000 rows in the table

regression=# INSERT into emp SELECT id, md5(random()::text), (SELECT city
FROM cities OFFSET id%25 LIMIT 1) FROM generate_series(1,10000) as id;

INSERT 0 10000

regression=# ANALYZE emp;

ANALYZE
```

Example

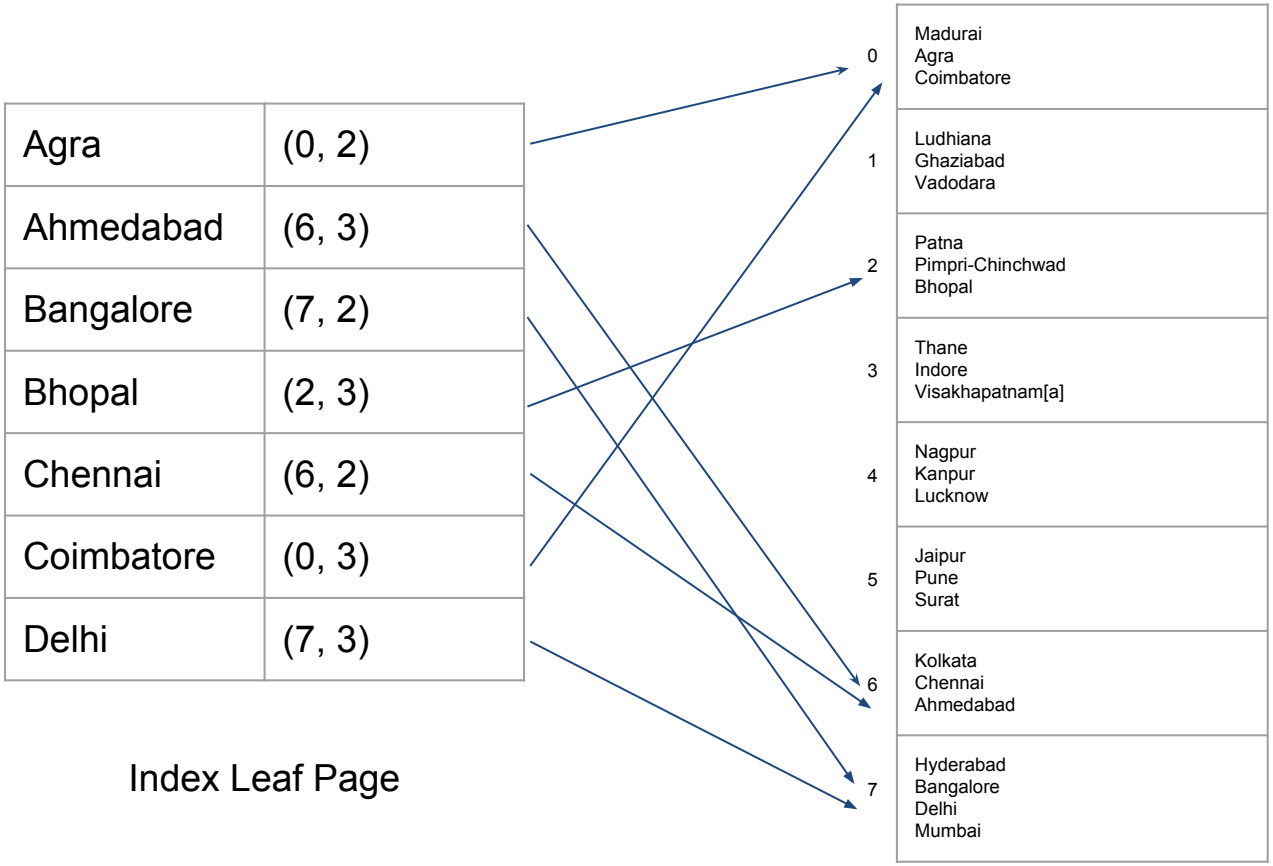
```
regression=# EXPLAIN SELECT * FROM emp WHERE id = 2;
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using pkindex on emp  (cost=0.29..8.30 rows=1 width=45)
   Index Cond: (id = 2)
(2 rows)
regression=# EXPLAIN SELECT * FROM emp WHERE city = 'Nagpur';
                                    QUERY PLAN
-------------------------------------------------------------------------------
-
 Bitmap Heap Scan on emp  (cost=11.38..111.39 rows=400 width=45)
   Recheck Cond: (city = 'Nagpur'::text)
   ->  Bitmap Index Scan on cityindex  (cost=0.00..11.29 rows=400 width=0)
         Index Cond: (city = 'Nagpur'::text)
(4 rows)
```

B-Tree Internals

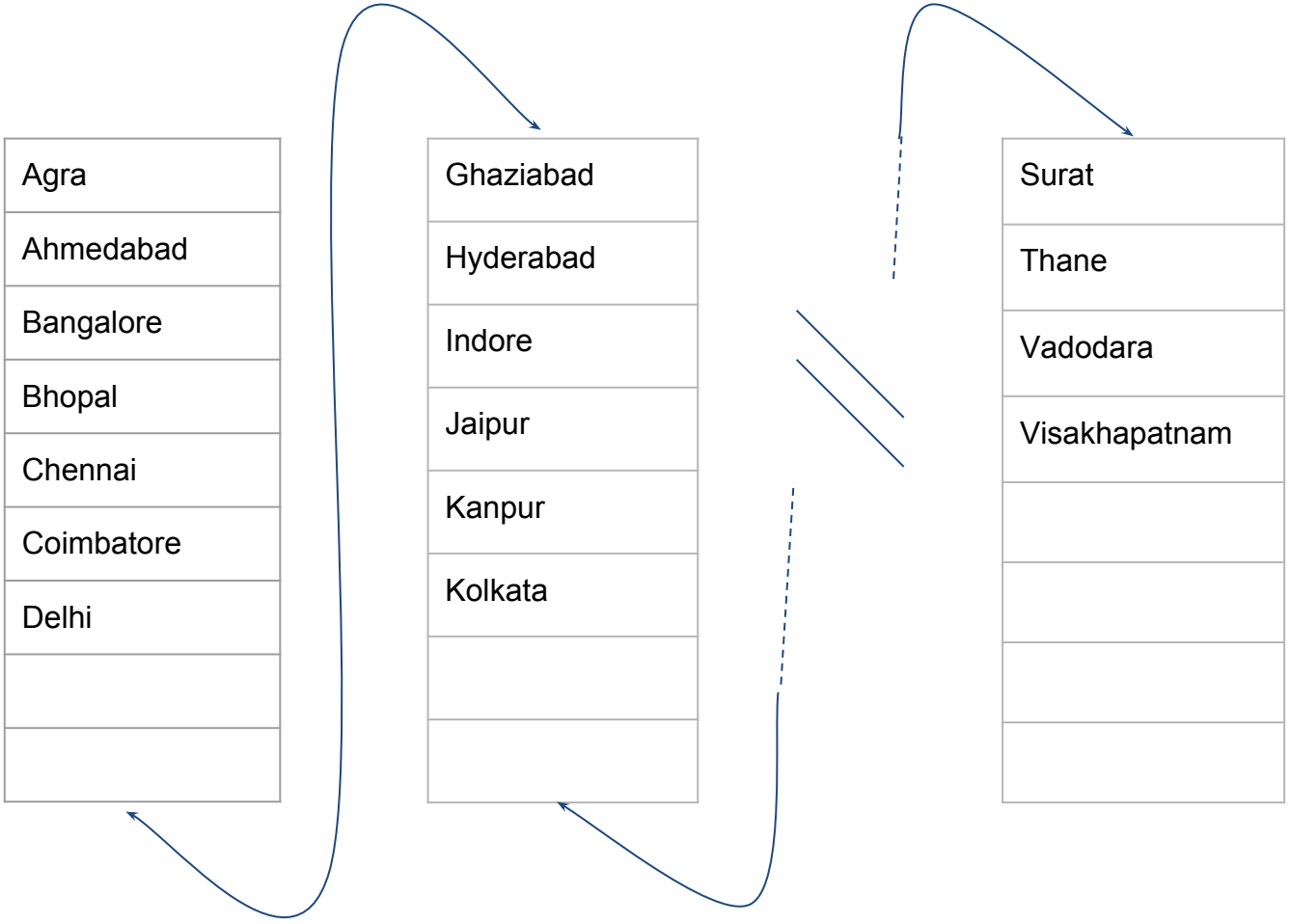
- Leaf pages store tuples consisting of `<key, TID>`, always ordered by key
- Leaf pages are linked by `next` and `prev` pointers
- Does not contain any visibility information

Agra	(0, 2)
Ahmedabad	(6, 3)
Bangalore	(7, 2)
Bhopal	(2, 3)
Chennai	(6, 2)
Coimbatore	(0, 3)
Delhi	(7, 3)

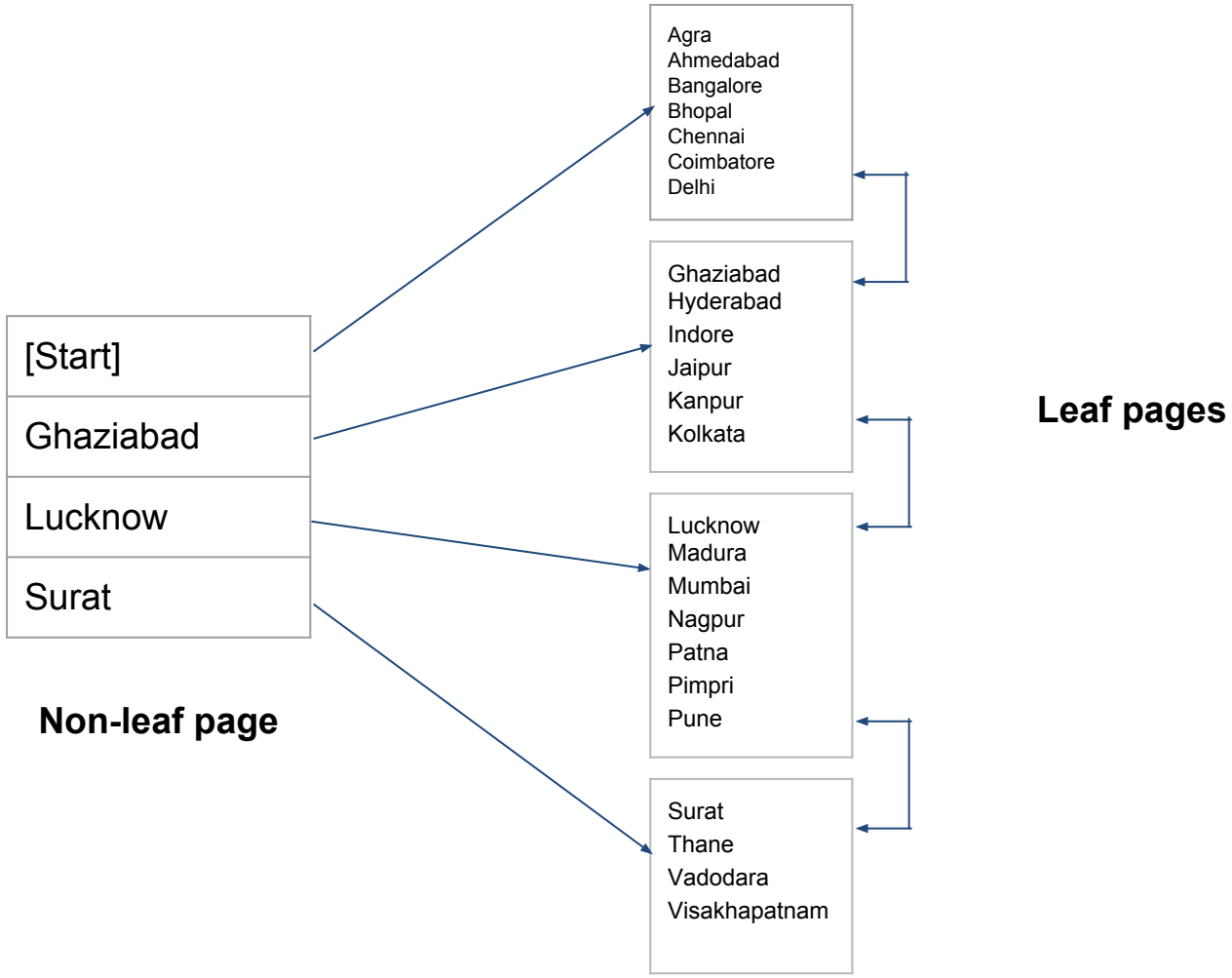
B-Tree Leaf Page



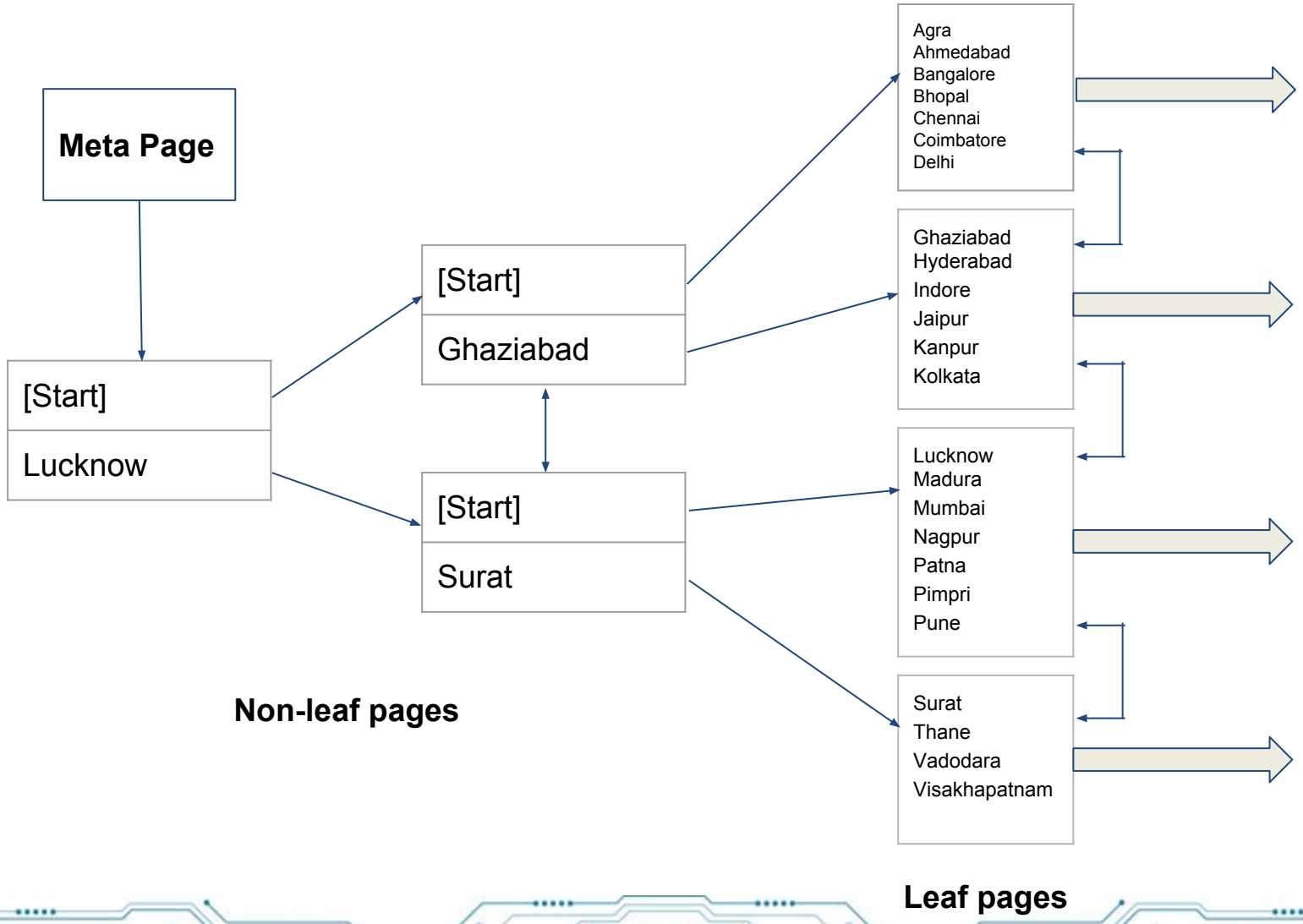
B-Tree Leaf Pages



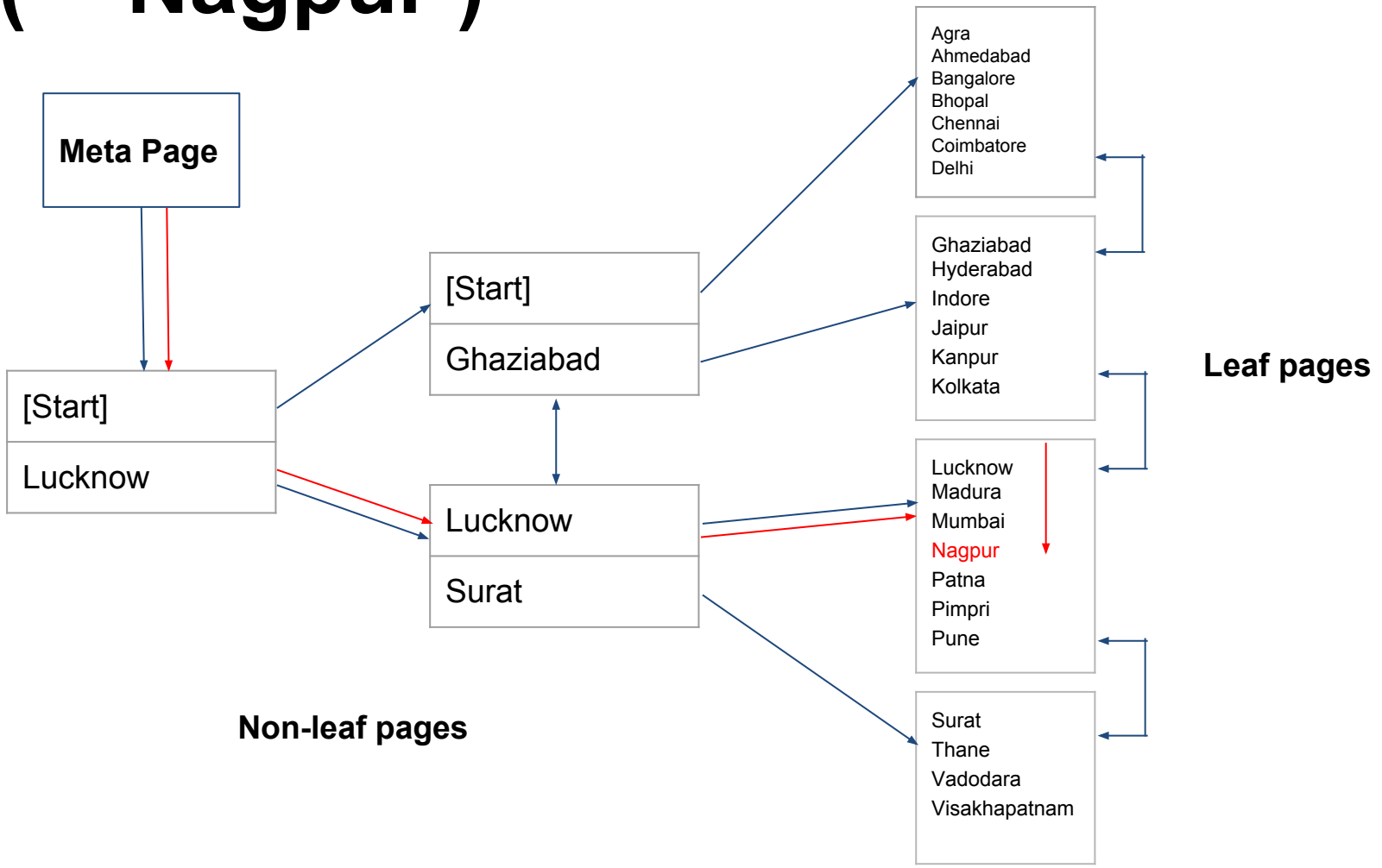
B-Tree Leaf Pages



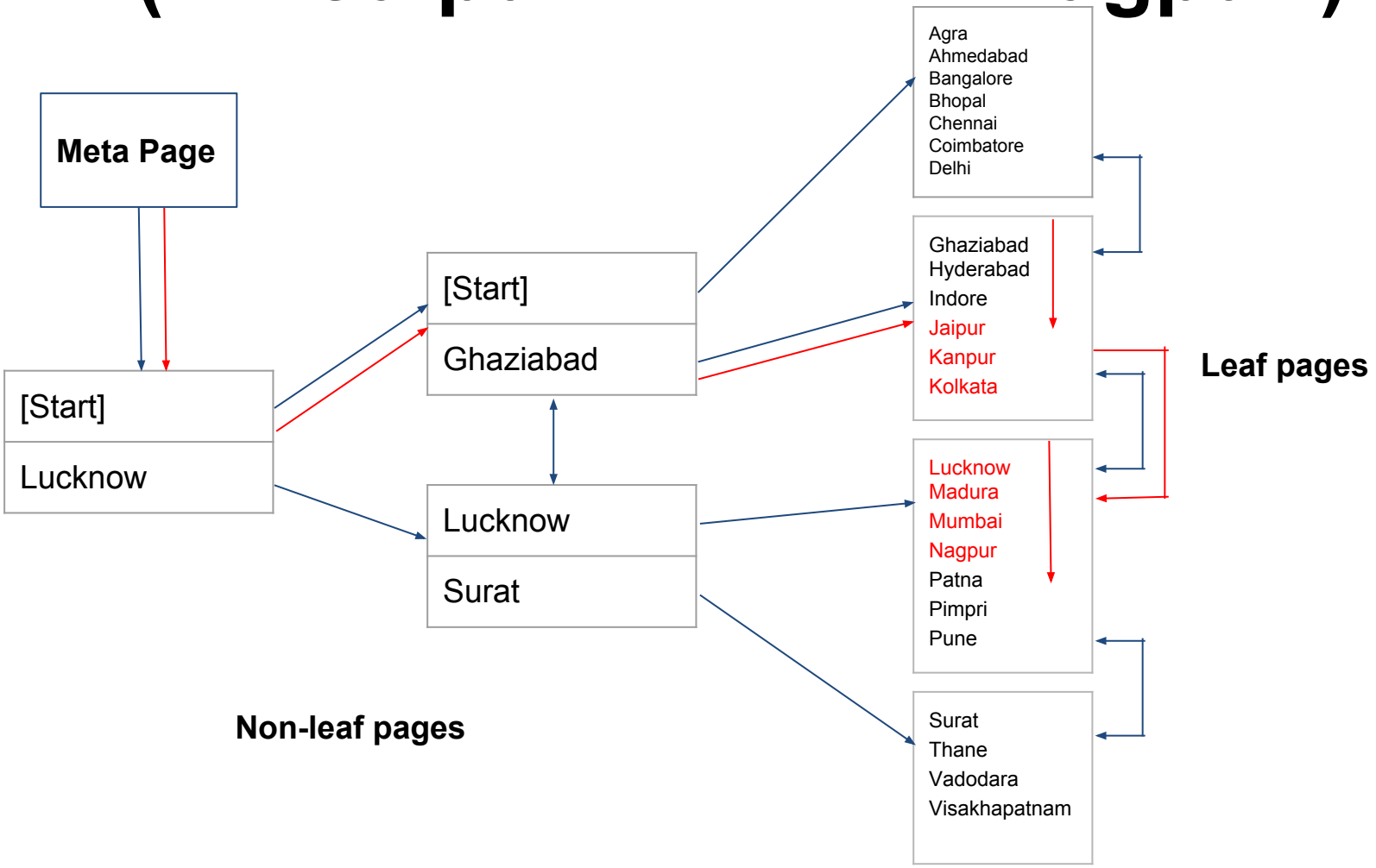
B-Tree Complete Picture



B-Tree Equality Search (= 'Nagpur')



B-Tree Range Search (>= ‘Jaipur’ AND <= ‘Nagpur’)



B-Tree Index

- The default index type in PostgreSQL
- WHERE key = K
- WHERE key >= K1 AND key <= K2
- Prefix matching LIKE queries (LIKE 'Am%')
- ORDER BY clause

Expression Indexes

```
regression=# EXPLAIN ANALYZE SELECT  * FROM emp
    WHERE lower(city) = 'nagpur';
                                                 QUERY PLAN
-------------------------------------------------------------------------------
-----------------------------
 Seq Scan on emp  (cost=0.00..245.00 rows=50 width=45) (actual
time=0.143..8.922 rows=400 loops=1)
   Filter: (lower(city) = 'nagpur'::text)
   Rows Removed by Filter: 9600
 Planning time: 0.138 ms
 Execution time: 8.999 ms
(5 rows)
```

Expression Indexes

```
regression=# DROP INDEX cityindex ;
DROP INDEX
regression=# CREATE INDEX cityindex ON emp(lower(city));
CREATE INDEX
regression=# ANALYZE emp;
ANALYZE
regression=# EXPLAIN ANALYZE SELECT  * FROM emp
    WHERE lower(city) = 'nagpur';
                                                        QUERY PLAN
-------------------------------------------------------------------------------
-----------------------------------------------
 Bitmap Heap Scan on emp  (cost=11.38..112.39 rows=400 width=45)
(actual time=0.133..0.412 rows=400 loops=1)
   Recheck Cond: (lower(city) = 'nagpur'::text)
   Heap Blocks: exact=95
   ->  Bitmap Index Scan on cityindex  (cost=0.00..11.29 rows=400
width=0) (actual time=0.110..0.110 rows=400 loops=1)
         Index Cond: (lower(city) = 'nagpur'::text)
 Planning time: 0.324 ms
 Execution time: 0.482 ms
(7 rows)
```

Expression Indexes

- User-defined functions, math functions, string functions
- Only `IMMUTABLE` functions are allowed
- The `WHERE` clauses must use the same expressions to take advantage of the index

Partial Indexes

- What if you want to index only some rows in the table?
- PostgreSQL allows you to specify `WHERE` clause in index definition to filter out uninteresting rows
- Shallow/smaller index, thus speed up `SELECT` queries
- Less overhead in `INSERT/UPDATE`

Partial Indexes

```
regression=# DROP INDEX cityindex ;
DROP INDEX
regression=# CREATE INDEX cityindex ON emp(city)
    WHERE city IN ('Delhi', 'Mumbai', 'Chennai', 'Kolkata');
CREATE INDEX
regression=# ANALYZE emp;
ANALYZE
regression=# EXPLAIN SELECT city FROM emp  WHERE city = 'Nagpur';
QUERY PLAN
-----------------------------------------------------
 Seq Scan on emp  (cost=0.00..220.00 rows=400 width=8)
   Filter: (city = 'Nagpur'::text)
(2 rows)
regression=# EXPLAIN SELECT city FROM emp  WHERE city = 'Mumbai';
                                  QUERY PLAN
-------------------------------------------------------------------------
-------
 Index Scan using cityindex on emp  (cost=0.28..15.28 rows=400 width=8)
   Index Cond: (city = 'Mumbai'::text)
(2 rows)
```

Index-Only Scans

```
regression=# EXPLAIN SELECT  id FROM emp WHERE id = 2;
                                QUERY PLAN
-------------------------------------------------------------------------
--
 Index Only Scan using pkindex on emp  (cost=0.29..8.30 rows=1
width=4)
   Index Cond: (id = 2)
(2 rows)
```

- Index scan should be cheaper than the sequential scan
- All required columns should be available in the index
- Visibility information should be up to date (requires VACUUM)

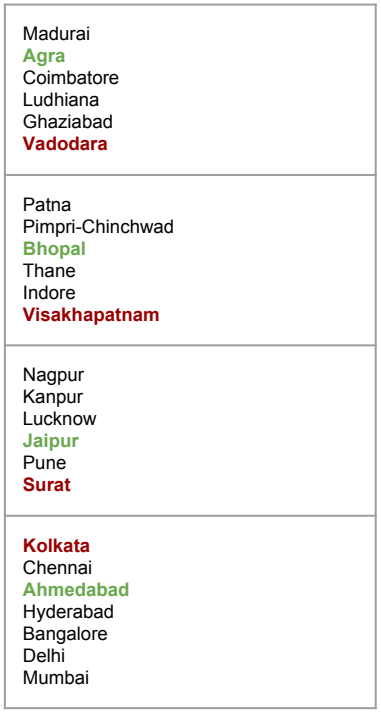
BRIN Indexes
(Block Range Index)

BRIN Index

- Useful for very large tables in which certain columns have some natural correlation with their physical location within the table e.g. insertion timestamp or sequence numbers
- Summary information (min/max) for a block or a range of blocks is stored in the index
- Small footprint, heap TIDs are not stored, so always lossy and hence SELECTs are slower (recheck needed)
- No VACUUM overhead

BRIN Index

Block 0	Agra, Vadodara
Block 1	Bhopal, Visakhapatnam
Block 2	Jaipur, Surat
Block 3	Ahmedabad, Kolkata



- Queries looking for `city = 'Vadodara'` can skip blocks 2 and 3 completely
- But queries looking for `city = 'Jaipur'` must scan all blocks

BRIN Index

Block 0	Agra, Coimbatore
Block 1	Delhi, Kolkata
Block 2	Lucknow, Patna
Block 3	Pimpri-Chinchwad, Visakhapatnam

Agra
Ahmedabad
Bangalore
Bhopal
Chennai
Coimbatore

Delhi
Ghaziabad
Hyderabad
Indore
Jaipur
Kanpur
Kolkata

Lucknow
Ludhiana
Madurai
Mumbai
Nagpur
Patna

Pimpri-Chinchwad
Pune
Surat
Thane
Vadodara
Visakhapatnam

- Clustering is important!
- Queries looking for `city = 'Jaipur'` can also skip 0, 2 and 3

GIN Indexes
(Generalized Inverted Index)

GIN Index

- The index stores a set of *(key, posting list)* where *posting list* is a set of sorted TIDs of the heap tuples where the key appears
- Same TID may appear in multiple index entries
- Useful when there are many duplicates

GIN Index

- For example:
  - GIN can be used to index documents and then search for keywords
  - Each word (key) along with the list of tuples (posting list) where it appears, is stored in the index
- `hstore, intarray, pg_trgm` are few examples where GIN indexes are helpful

Hash Indexes

Hash Index

- Only supports equality queries
- Crash safe starting PG 10
- Replication support starting PG 10
- Hash indexes can be smaller in size for very large keys since only hash values are stored in the index, but collisions can make `SELECT` slower

Hash Index

```
regression=# CREATE INDEX btree_empname ON emp(empname);
CREATE INDEX
regression=# CREATE INDEX hash_empname ON emp USING HASH(empname);
CREATE INDEX

regression=# \di+ btree_empname
List of relations
 Schema |     Name      | Type  | Owner | Table |  Size   |
Description
--------+---------------+-------+-------+-------+---------+-----------
--
 public | btree_empname | index | pavan | emp   | 4072 kB |
(1 row)

regression=# \di+ hash_empname
                          List of relations
 Schema |     Name     | Type  | Owner | Table |  Size  | Description
--------+--------------+-------+-------+-------+--------+-------------
 public | hash_empname | index | pavan | emp   | 528 kB |
(1 row)
```

GiST Indexes
(Generalized Search Trees)

GiST Index

- Stands for Generalized Search Trees
- No order within pages
- A tuple can be stored in any leaf page as long as the higher level page store correct information
  - For points, a bounding box of all points below it
  - For intarray, the OR of all the nodes below it

GiST Index Use Cases

- GIS queries
- Bounding box queries
- Nearest neighbours

Evaluating Indexing Strategy

- PostgreSQL collects several stats to detect potential lack of indexes or unused indexes
  - `pg_stat_user_tables`
  - `pg_stat_user_indexes`
- `pg_stat_statements` and `pg_auto_explain` extensions can be used to find problem queries (there are others too)
- Again, sequential scans are not always bad!

Finding Missing Index

```
$ pgbench -p 5433 -T 30 -c 1 regression
<snip>
number of transactions actually processed: 108
latency average = 279.961 ms
tps = 3.571921 (including connections establishing)
```

- That's too bad!
- Start by looking at the stats (and slow queries logged somewhere)

Finding Missing Index

```
regression=# select relname, seq_scan, seq_tup_read, idx_scan from
pg_stat_user_tables ;
     relname      | seq_scan | seq_tup_read | idx_scan
------------------+----------+--------------+----------
 pgbench_tellers  |      109 |        10900 |        0
 pgbench_history  |        0 |            0 |
 pgbench_accounts |      434 |    217000000 |
 pgbench_branches |      110 |         1100 |        0
(4 rows)
```

- None of the tables use index scans, but `pgbench_accounts` stands out because of 500000 tuple reads per sequential scan.

Finding Missing Index

- Ah! Found the culprit!
- Someone dropped the much needed primary key

```
regression=# ALTER TABLE pgbench_accounts DROP CONSTRAINT
pgbench_accounts_pkey;
```

- Let's fix it!

```
regression=# ALTER TABLE pgbench_accounts ADD PRIMARY KEY (aid);
ALTER TABLE
$ pgbench -p 5433 -T 30 -c 1 regression
<snip>
number of transactions actually processed: 22008
tps = 733.586903 (including connections establishing)
```

Finding Unused Indexes

- Unused indexes take disk space
- Slow down INSERT/UPDATE/VACUUM
- How to find them?

```
regression=# CREATE INDEX pa_balance ON pgbench_accounts
(abalance);

CREATE INDEX


$ pgbench -p 5433 -T 30 -c 1 regression
<snip>
tps = 686.119003 (including connections establishing)
```

Finding Unused Indexes

```
regression=# SELECT relname, seq_scan, idx_scan
                FROM pg_stat_user_tables ;
     relname      | seq_scan | idx_scan
------------------+----------+----------
 pgbench_accounts |        4 |    41172
 pgbench_history  |        0 |
 pgbench_tellers  |    20587 |        0
 pgbench_branches |    20588 |        0
(4 rows)

regression=# SELECT indexrelname, idx_scan
                FROM pg_stat_user_indexes ;
     indexrelname      | idx_scan
-----------------------+----------
 pgbench_branches_pkey |        0
 pgbench_tellers_pkey  |        0
 pgbench_accounts_pkey |    41172
 pa_balance            |        0
(4 rows)
```

Summary

- B-Tree indexes for equality and range queries
- Hash indexes for equality
- BRIN indexes for very large data, support equality and range, but lossy and slow
- GIN when a column has many duplicates, multiple keys per tuple
- GiST for bounding box, nearest neighbours and everything else

Summary

- Analyse slow queries and look for any missing indexes. But remember all sequential scans are not bad
- Unused indexes add to INSERT/UPDATE costs. But beware of Hot Standbys
- VACUUM can help index-only scans

2ndQuadrant® PostgreSQL

We're Hiring!

- **Sustainable** Open Source Development
- 24/7 Bug Fix Support
- On-premise, Cloud, Hybrid with RemoteDBA
- Consulting & Training