Postgres Performance for Humans

@craigkerstiens

heroku

Shameless plugs

http://www.postgresweekly.com

http://www.craigkerstiens.com

http://www.postgresguide.com

http://www.postgresapp.com

http://www.heroku.com/postgres

Postgres - TLDR

Postgres - TLDR

Datatypes
Conditional Indexes
Transactional DDL
Foreign Data Wrappers
Concurrent Index Creation
Extensions
Common Table Expressions
Fast Column Addition
Listen/Notify
Table Inheritance
Per Transaction sync replication
Window functions
NoSQL inside SQL
Momentum

TLDR in a quote

"It's the emacs of databases"

http://www.craigkerstiens.com/2012/04/30/why-postgres/

OLTP vs OLAP

OLTP *vs* OLAP

Web apps

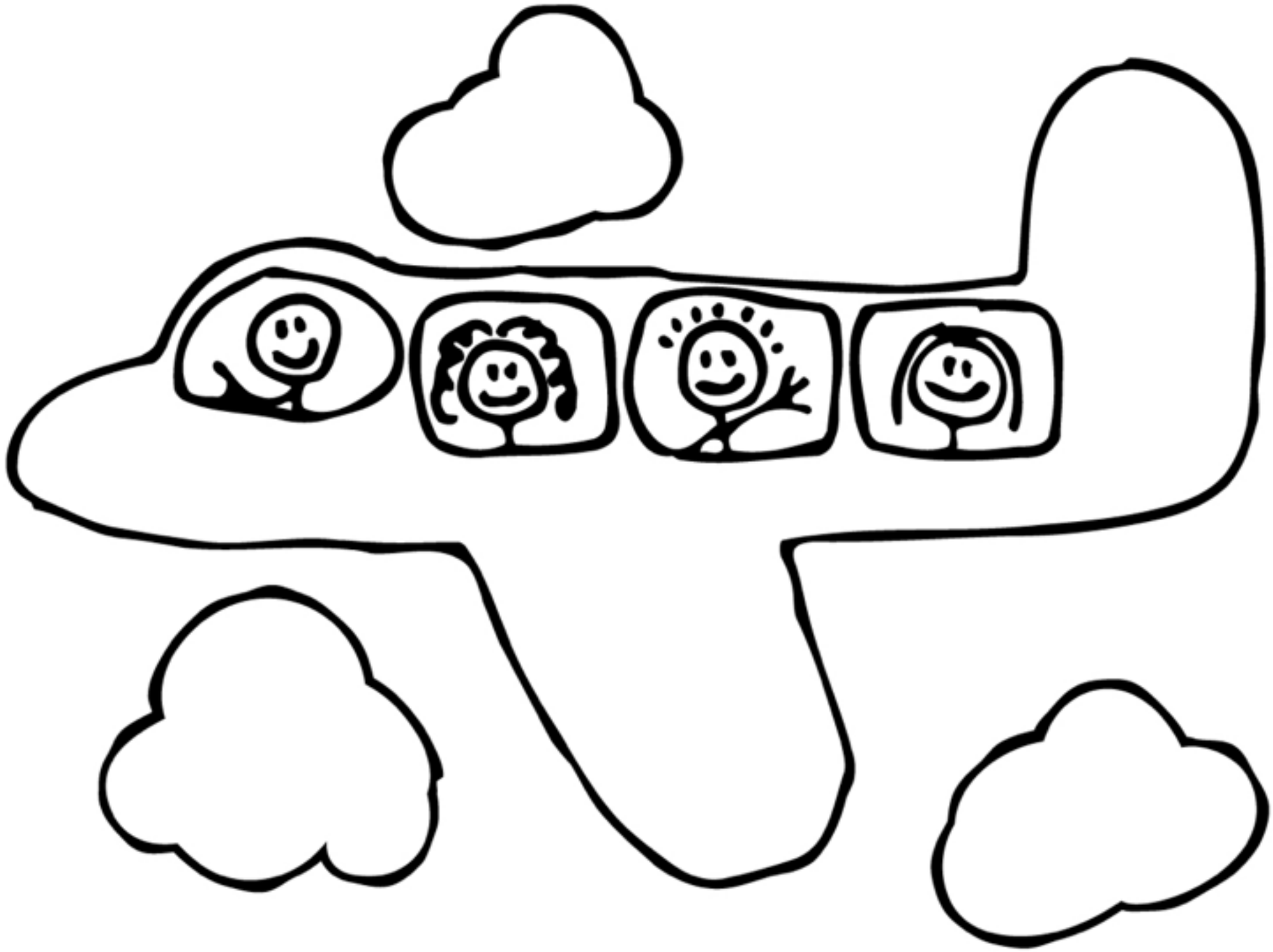
OLTP vs OLAP

BI/Reporting

Postgres Setup/Config

- On Amazon
  - Use Heroku OR 'postgresql when its not your dayjob'
- Other clouds
  - 'postgresql when its not your dayjob'
- Real hardware
  - High performance PostgreSQL

http://thebuild.com/blog/2012/06/04/postgresql-when-its-not-your-job-at-djangocon-europe/



Cache rules everything around me

Cache Hit Rate

```
SELECT
        'index hit rate' as name,
        (sum(idx_blks_hit) - sum(idx_blks_read)) /
sum(idx_blks_hit + idx_blks_read) as ratio
      FROM pg_statio_user_indexes
      union all
      SELECT
       'cache hit rate' as name,
        case sum(idx_blks_hit)
          when 0 then 'NaN'::numeric
          else to_char((sum(idx_blks_hit) -
sum(idx_blks_read)) / sum(idx_blks_hit + idx_blks_read),
'99.99')::numeric
        end as ratio
     FROM pg_statio_user_indexes)
```

Cache Hit Rate

```
      name       |          ratio
-----------------+-------------------------
 cache hit rate |                      0.99
```

Index Hit Rate

```
SELECT
  relname,
  100 * idx_scan / (seq_scan + idx_scan),
  n_live_tup
FROM pg_stat_user_tables
ORDER BY n_live_tup DESC;
```

Index Hit Rate

relname	percent_of_times_index_used	rows_in_table
events	0	669917
app_infos_user_info	0	198218
app_infos	50	175640
user_info	3	46718
rollouts	0	34078
favorites	0	3059
schema_migrations	0	2
authorizations	0	0
delayed_jobs	23	0

Rough guidelines

Cache hit rate >= 99%

Index hit rate >= 95%

where on > 10,000 rows

Shortcuts

psql

```
$ cat ~/.psqlrc

\set ON_ERROR_ROLLBACK interactive

-- automatically switch between extended and normal
\x auto

-- always show how long a query takes
\timing
```

```
\set show_slow_queries
'SELECT
  (total_time / 1000 / 60) as total_minutes,
  (total_time/calls) as average_time, query
FROM pg_stat_statements
ORDER BY 1 DESC
LIMIT 100;'
```

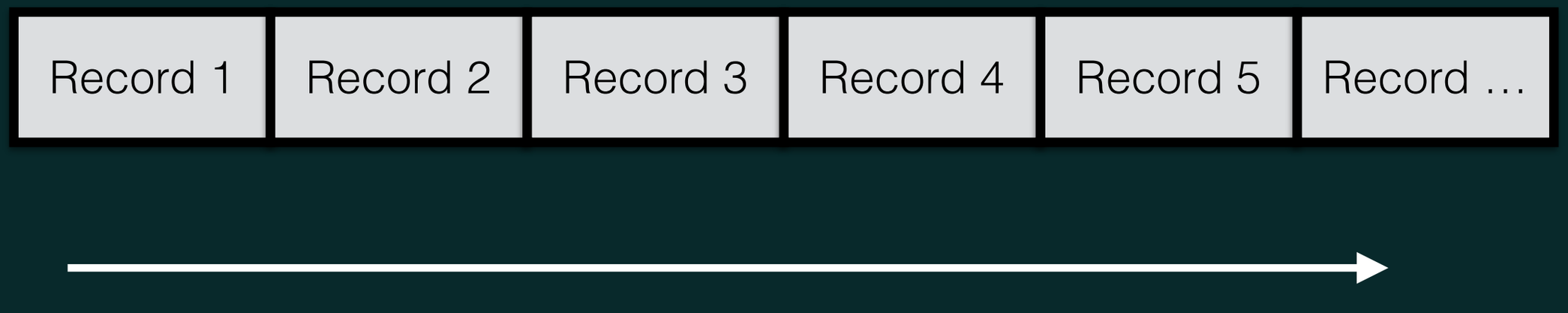
psql

```
$ cat ~/.psqlrc

\set ON_ERROR_ROLLBACK interactive

-- automatically switch between extended and normal
\x auto

-- always show how long a query takes
\timing
```

```
\set show_slow_queries
'SELECT
  (total_time / 1000 / 60) as total_minutes,
  (total_time/calls) as average_time, query
FROM pg_stat_statements
ORDER BY 1 DESC
LIMIT 100;'
```

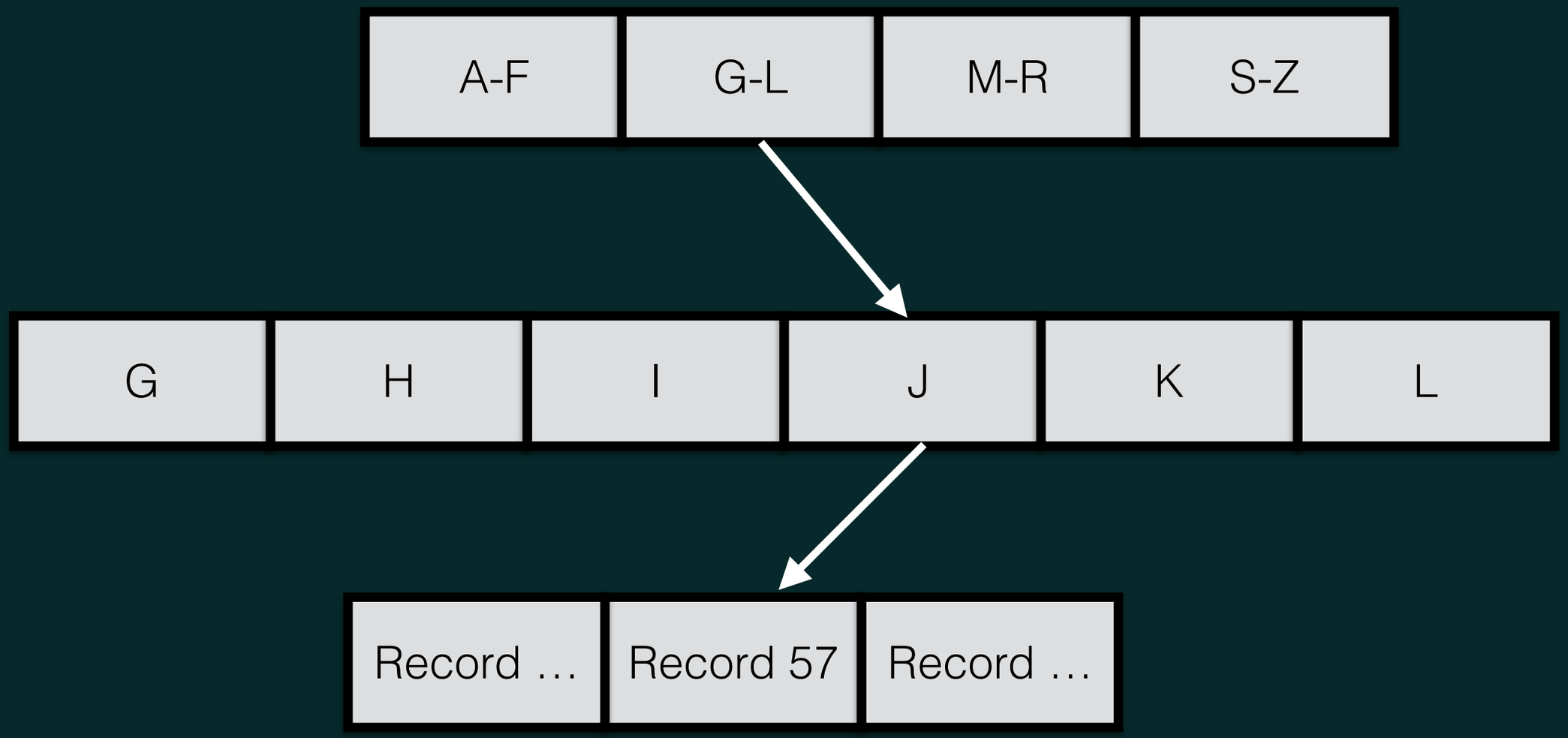
How Data is Retrieved

Sequential Scanning



Record 1
Record 2
Record 3
Record 4
Record 5
Record …

Index Scans



Sequential Scans

Good for large reports

Computing over lots of data (1k + rows)

Index Scans

Good for small results

Most common queries in your app

Understanding Specific Query Performance

Understanding Query Performance

```
SELECT last_name
FROM employees
WHERE salary >= 50000;
```

Explain

```
# EXPLAIN
  SELECT last_name
  FROM employees
  WHERE salary >= 50000;

                        QUERY PLAN
-----------------------------------------------------------
Seq Scan on employees (cost=0.00..35811.00 rows=1
width=6)
  Filter: (salary >= 50000)
(3 rows)
```

Explain

```
# EXPLAIN
  SELECT last_name
  FROM employees
  WHERE salary >= 50000;
                    QUERY PLAN
 ------------------------------------------------------------
  Seq Scan on employees (cost=0.00..35811.00 rows=1
 width=6)
    Filter: (salary >= 50000)
 (3 rows)
```

startup time

max time

rows return

Explain Analyze

```
# EXPLAIN ANALYZE
  SELECT last_name
  FROM employees
  WHERE salary >= 50000;
                    QUERY PLAN
 ------------------------------------------------------------
Seq Scan on employees (cost=0.00..35811.00 rows=1
width=6)  (actual time=2.401..295.247 rows=1428
loops=1)
  Filter: (salary >= 50000)
Total runtime: 295.379
(3 rows)

  Filter: (salary >= 50000)
(3 rows)
```

startup time

max time

rows return

actual time

Rough guidelines

Page response times < 100 ms

Common queries < 10ms

Rare queries < 100ms

Explain Analyze

```
# EXPLAIN ANALYZE
  SELECT last_name
  FROM employees
  WHERE salary >= 50000;
                    QUERY PLAN
 ------------------------------------------------------------
Seq Scan on employees (cost=0.00..35811.00 rows=1
width=6)  (actual time=2.401..295.247 rows=1428
loops=1)
  Filter: (salary >= 50000)
Total runtime: 295.379
(3 rows)

  Filter: (salary >= 50000)
(3 rows)
```

startup time

max time

rows return

actual time

Indexes!

```
# CREATE INDEX idx_emps ON employees (salary);
```

Indexes!

```
EXPLAIN ANALYZE
  SELECT last_name
  FROM employees
  WHERE salary >= 50000;
                    QUERY PLAN
-----------------------------------------------------------
Index Scan using idx_emps on employees
(cost=0.00..8.49 rows=1 width=6) (actual time =
0.047..1.603 rows=1428 loops=1)
  Index Cond: (salary >= 50000)
Total runtime: 1.771 ms
(3 rows)
```

pg_stat_statements

$ select * from pg_stat_statements where query ~ 'from users where email';

```
userid              | 16384
dbid                | 16388
query               | select * from users where email = ?;
calls               | 2
total_time          | 0.000268
rows                | 2
shared_blks_hit     | 16
shared_blks_read    | 0
shared_blks_dirtied | 0
shared_blks_written | 0
local_blks_hit      | 0
local_blks_read     | 0
local_blks_dirtied  | 0
local_blks_written  | 0
...
```

pg_stat_statements

```
SELECT
  (total_time / 1000 / 60) as total,
  (total_time/calls) as avg,
  query
FROM pg_stat_statements
ORDER BY 1 DESC
LIMIT 100;
```

pg_stat_statements

```
 total   |    avg   |              query
---------+----------+---------------------------
 295.76  |   10.13  | SELECT id FROM users...
 219.13  |   80.24  | SELECT * FROM ...
(2 rows)
```

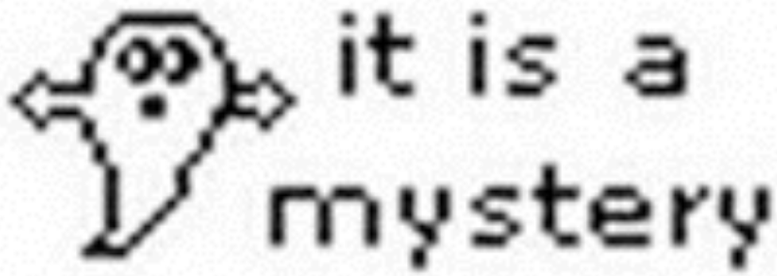
Indexes

Indexes

B-Tree
Generalized Inverted Index (GIN)
Generalized Search Tree (GIST)
K Nearest Neighbors (KNN)
Space Partitioned GIST (SP-GIST)

Indexes

Which do I use?



BTree

This is what you usually want

Generalized Inverted Index (GIN)

Use with multiple values in 1 column

Array/hStore

Generalized Search Tree (GIST)

- Full text search
- Shapes

Indexes

B-Tree
Generalized Inverted Index (GIN)
Generalized Search Tree (GIST)
K Nearest Neighbors (KNN)
Space Partitioned GIST (SP-GIST)
VODKA (Coming soon)

More indexes

Indexes

Conditional

Functional

Concurrent creation

Conditional

```
> SELECT *
FROM places;
```

```
    name     |      population
-------------------------------
ACMAR        |     6055
ARAB         |    13650
```

Conditional

```
> SELECT *
FROM places
WHERE population > 10000;
```

```
    name     |      population
-------------------------------------
ARAB         |   13650
```

Conditional

```
> CREATE INDEX idx_large_population ON
places(name) where population > 10000;
```

Functional

```
> SELECT *
FROM places;
```

```
                  data
-------------------------------------
{"city": "ACMAR", "pop": 6055}
{"city": "ARAB", "pop": 13650}
```

Functional

```
> SELECT *
FROM places
WHERE get_numeric('pop', data) > 10000;
```

```
                  data
-------------------------------------
{"city": "ARAB", "pop": 13650}
```

Functional

```
> CREATE INDEX idx_large_population ON
places(get_numeric('pop', data));
```

Conditional and Functional

```
> CREATE INDEX idx_large_population ON
places(data) WHERE
get_numeric('pop', data) > 10000;
```

One more thing

CREATE INDEX CONCURRENTLY ...

roughly 2-3x slower
Doesn't lock table

hstore / JSON

hstore

```
CREATE EXTENSION hstore;
CREATE TABLE users (
    id integer NOT NULL,
    email character varying(255),
    data hstore,
    created_at timestamp without time zone,
    last_login timestamp without time zone
);
```

hstore

```
INSERT INTO users
VALUES (
 1,
 'craig.kerstiens@gmail.com',
 'sex => "M", state => “California”',
 now(),
 now()
);
```

JSON

```
SELECT
     '{"id":1,"email":
       "craig.kerstiens@gmail.com",}'::json;
```

hstore

Indexes work

gin

gist

json

Functional indexes work

have fun

jsonb

The world is better

Pooling

Premier Pools & Spas

Options

Application/Framework layer

Stand alone daemon

PG options

pgbouncer
pgpool

Adding Cache

Replication options

slony

londiste

bucardo

pgpool

wal-e

barman

Replication options

slony

londiste

bucardo

pgpool

wal-e

barman

Backups

Logical

pg_dump

can be human readable, is portable

Physical

The bytes on disk

Base backup

Logical

Good across architectures

Good for portability

Has load on DB

Works < 50 GB

Physical

More initial setup

Less portability

Limited load on system

Use above 50 GB

Recap

Recap

OLAP

Whole other talk
Disk IO is important
Order on disk is helpful (pg-reorg)
MPP solutions on top of Postgres

Recap

OLTP (webapps)

Ensure bulk of data is cache

Optimize overall query load with pg_stat_statements

Efficient use of indexes

When cache sucks, throw more at it

Questions

http://www.speakerdeck.com/u/craigkerstiens/