How to Reason About Relational Database Performance

Data is stored in rows.

```
id | name | email | phone | password | ...
---+------+-------+-------+----------+----
 1 | Bob  | b@... | 703...| ???      | ...
 2 | Jan  | j@... | 812...| ???      |
 3 | Ed   | e@... | ...
 4 | Joe  | jo@.. | ...
```

Rows have columns of different types.

Table 8-2. Numeric Types

Name	Storage Size	Description	Range
smallint	2 bytes	small-range integer	-32768 to +32767
integer	4 bytes	typical choice for integer	-2147483648 to +2147483647
bigint	8 bytes	large-range integer	-9223372036854775808 to 9223372036854775807
decimal	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
numeric	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
real	4 bytes	variable-precision, inexact	6 decimal digits precision
double precision	8 bytes	variable-precision, inexact	15 decimal digits precision
serial	4 bytes	autoincrementing integer	1 to 2147483647
bigserial	8 bytes	large autoincrementing integer	1 to 9223372036854775807

http://www.postgresql.org/docs/9.1/static/datatype-numeric.html

Table 8-4. Character Types

Name	Description
character varying(*n*), varchar(*n*)	variable-length with limit
character(*n*), char(*n*)	fixed-length, blank padded
text	variable unlimited length

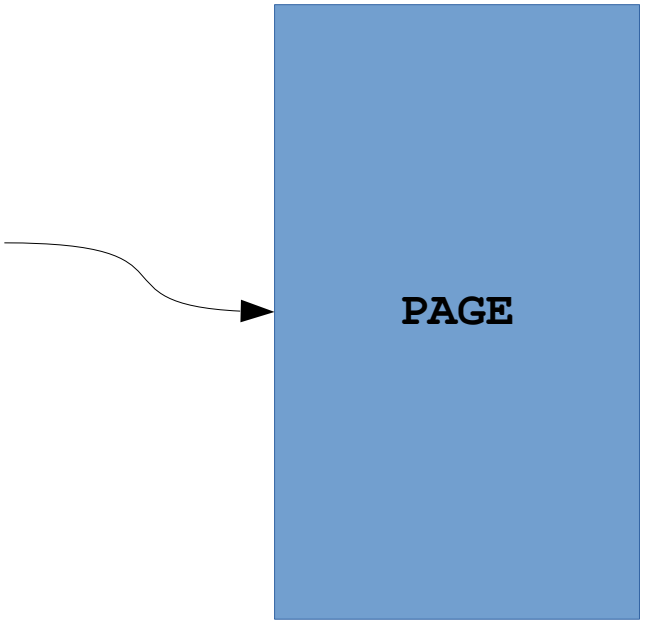
http://www.postgresql.org/docs/9.1/static/datatype-character.html

Null fields require an extra bit of storage.

Big fields (> 2kb) are stored outside of the page.
This is called TOAST. (The Oversized-Attribute Storage Technique)
http://www.postgresql.org/docs/9.4/static/storage-toast.html

Rows are stored in a page.

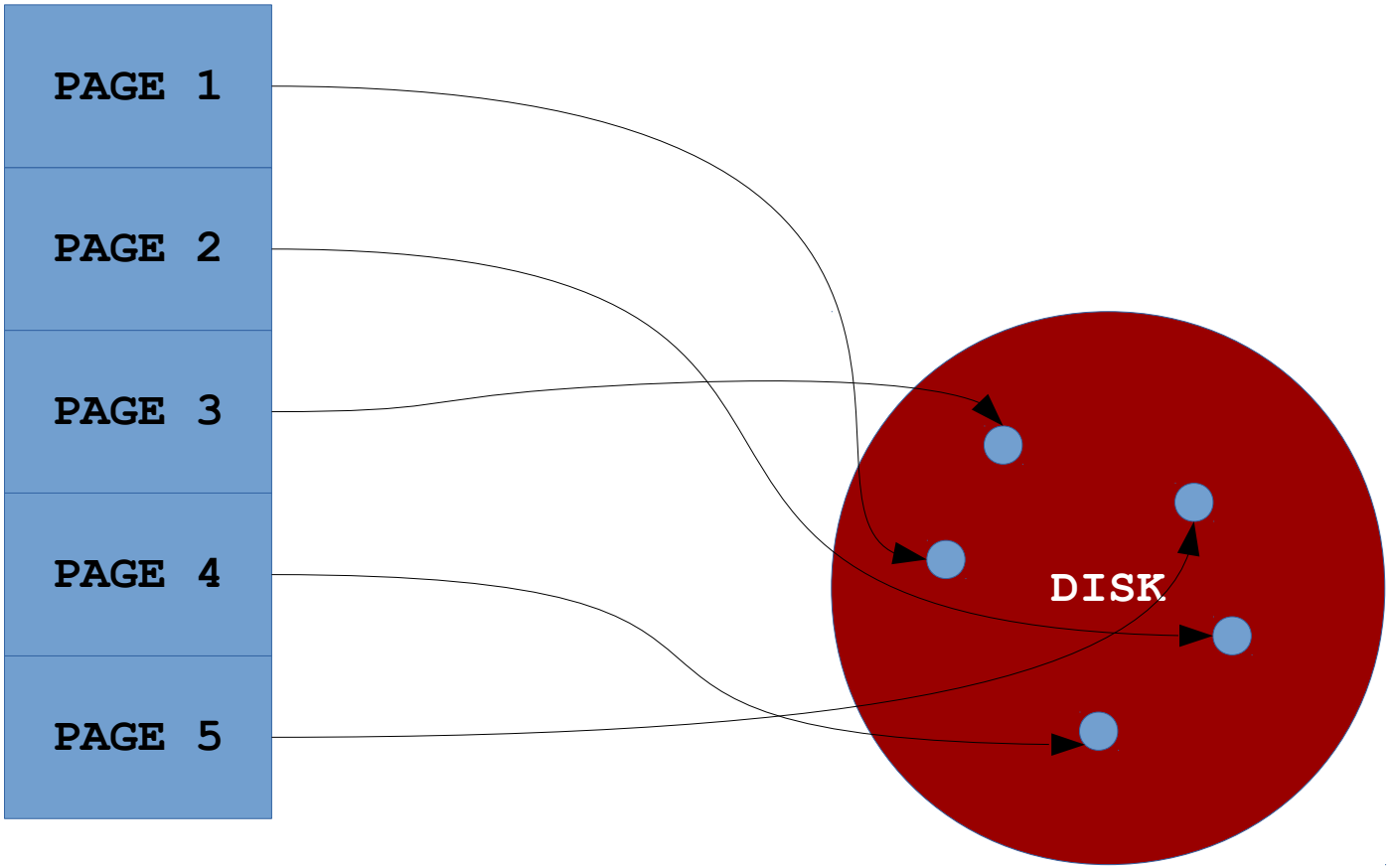
```
id | name | email | phone | password | ...
---+------+-------+-------+----------+----
 1 | Bob  | b@... | 703...| ???      | ...
 2 | Jan  | j@... | 812...| ???      |
 3 | Ed   | e@... | ...
 4 | Joe  | jo@.. | ...
```



Pages are typically 8kb in size.

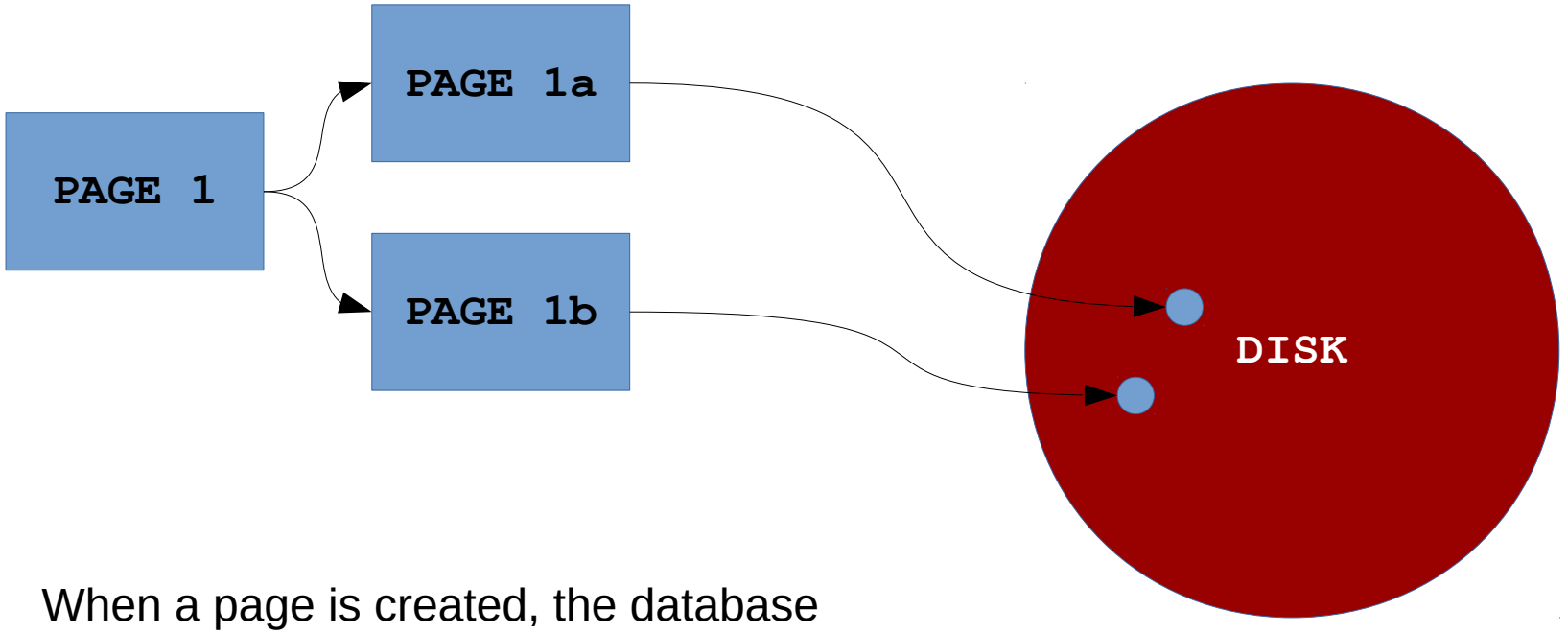
More Info: http://www.postgresql.org/docs/9.3/static/storage-page-layout.html

Pages are stored on disk.



More Info: http://www.postgresql.org/docs/9.3/static/storage-file-layout.html

When a page fills up, it gets split.



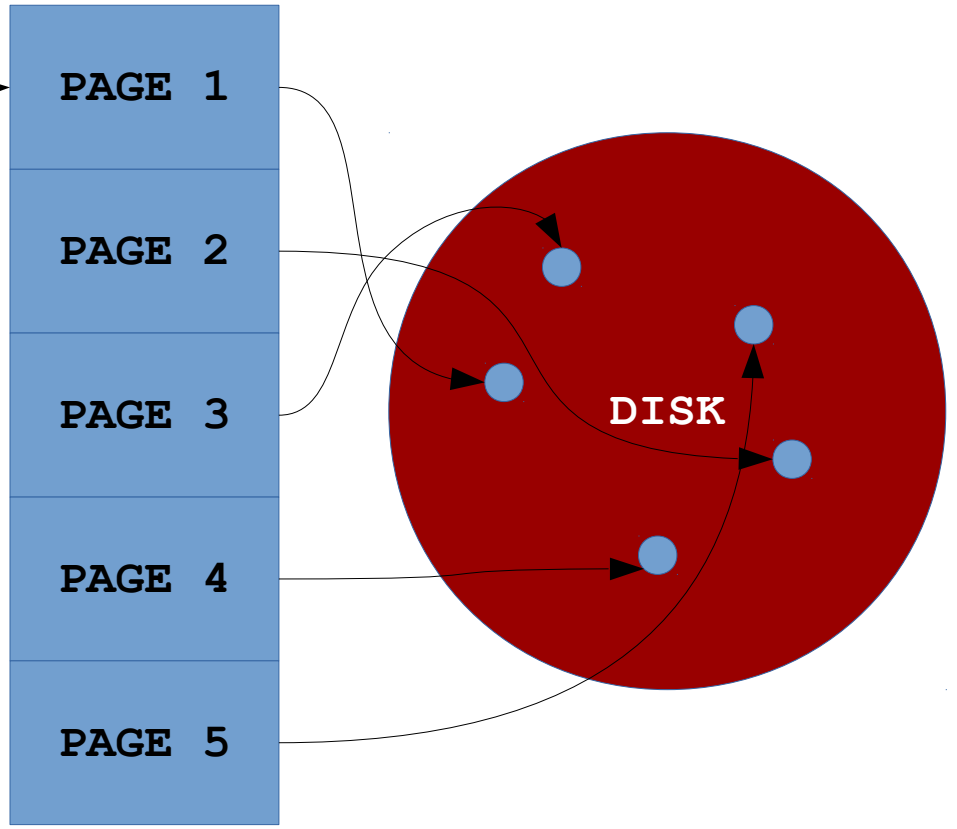
When a page is created, the database tries to leave some extra space so that values can change without triggering a split. This is controlled by a “fill factor” setting.

Thought Experiments

- How many 50 byte rows can a page hold?
  How many 500 byte rows?
  How many 1kb rows?
- You query a table with big rows (~1kb) but you only need one or two fields. The query returns 10,000 rows.
  How many pages does the database read?
- Same query, except the rows are small (~50 bytes).
  How many pages does the database read?
- Assume it takes ~10ms to read a page from the disk.
  How long do these queries take?

The 'primary key index' maps the primary key to a page.

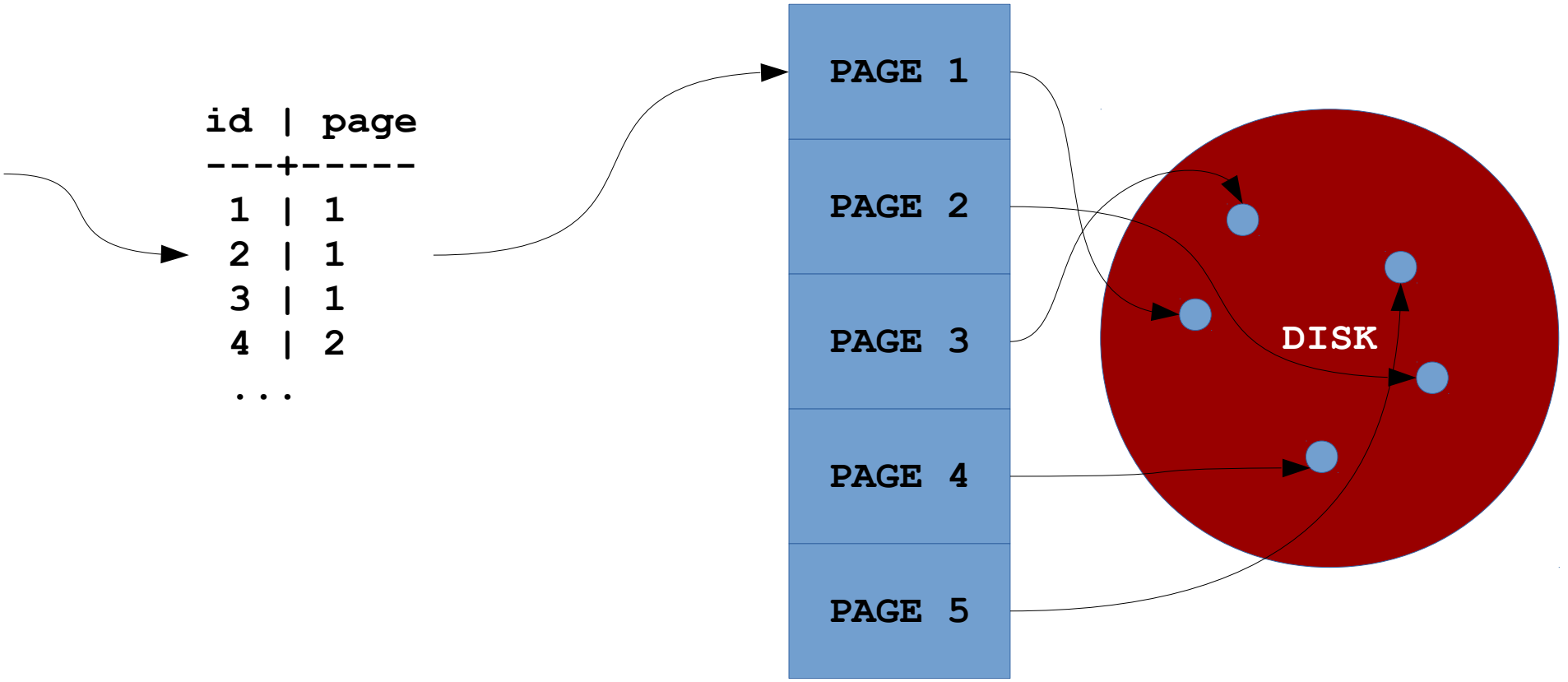
```
id | page
---+-----
 1 | 1
 2 | 1
 3 | 1
 4 | 2
 ...
```



This is cached in RAM.

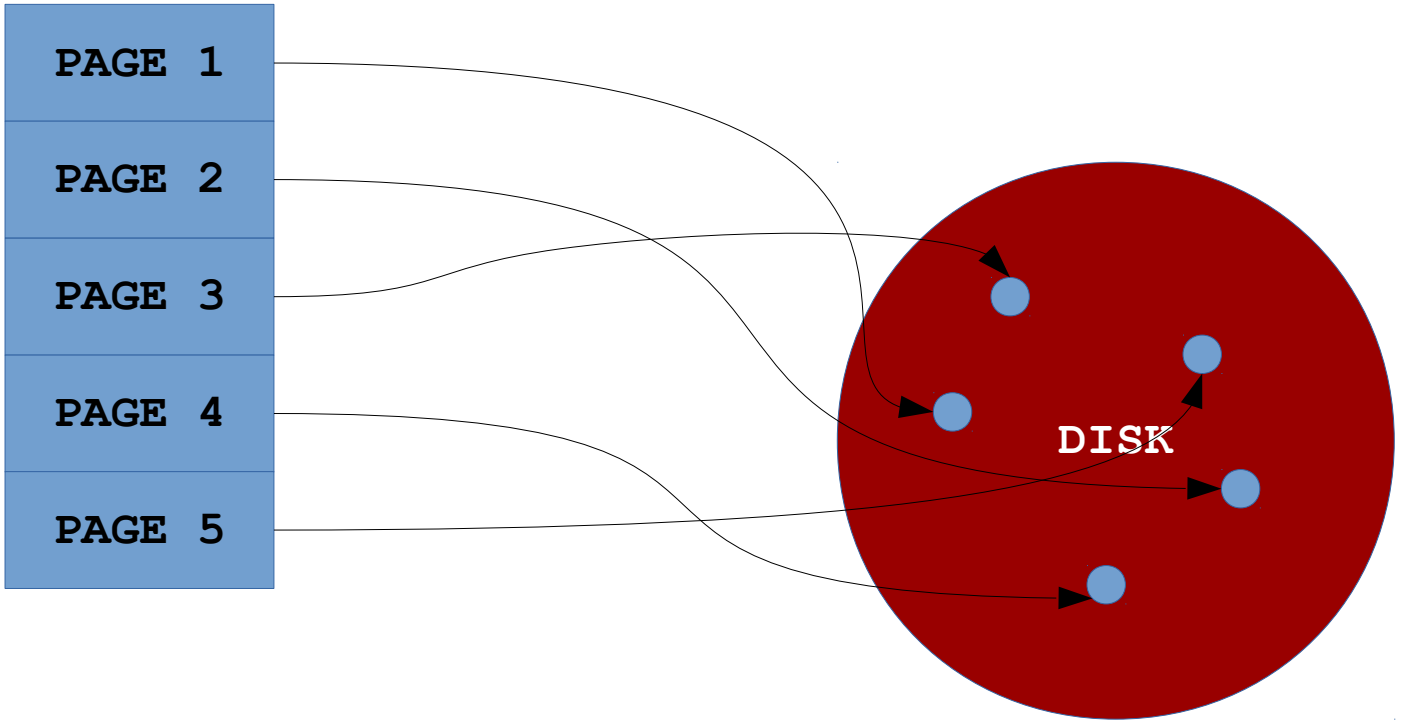
Primary key lookups are fast.
One RAM lookup, one disk read.

To fetch data efficiently, you must know the primary key.



What about queries on other fields?
What about joins?

If we don't know the primary key, we need to examine every row.



This is called a 'table scan' or 'sequence scan' and is expensive.

How expensive?

- Cost is:

```
(disk pages read * seq_page_cost) +
(rows scanned * cpu_tuple_cost)
```

  - Our prospects table has 65,909 pages and 1.9M rows.
  - seq_page_cost = 1
  - cpu_tuple_cost = 0.01
  - ```
    SELECT relpages, reltuples
    FROM pg_class WHERE relname = 'prospects';
    ```

- (65909 * 1 + (1.9*10^6) * 0.01) = **84,909**
- How does this translate to query time? Hard to tell. Depends on machine specs (disk speed, RAM, cores) and how much data is already cached in RAM.
    ```

This is why we have indexes.

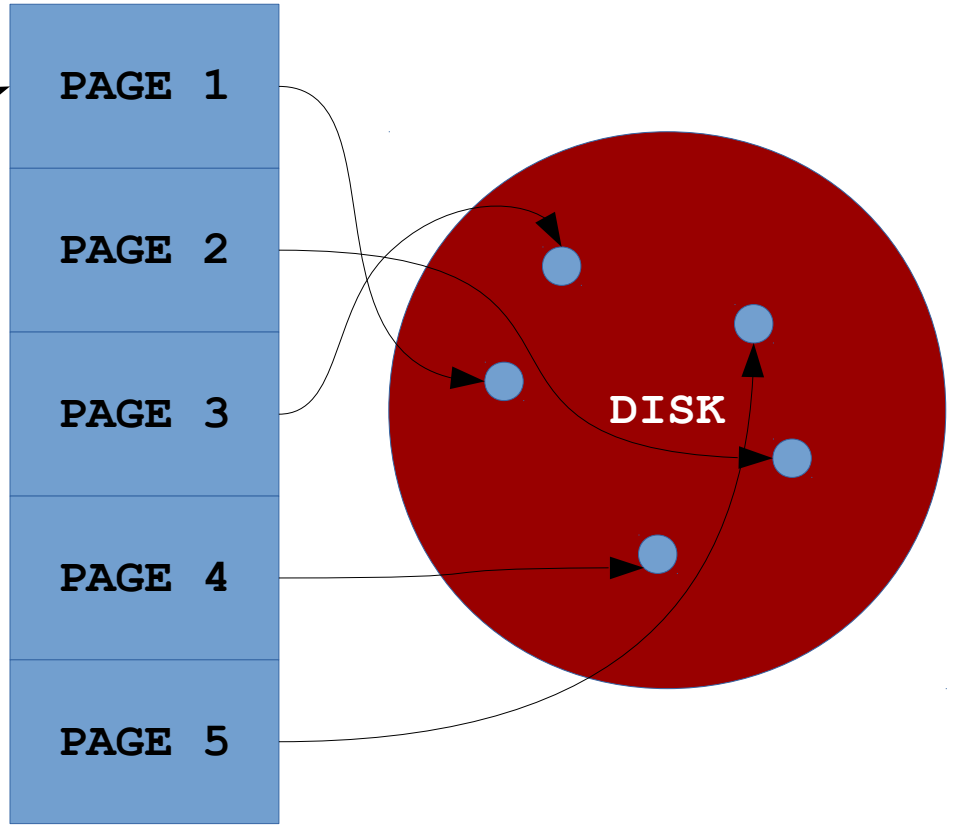
“All problems in computer science can be solved by another level of indirection.”

- Butler Lampson

An index maps a value to a primary key.

```
 foo_id   | id
----------+-----
 1        | 4
 2        | 8
 3        | 1
 4        | 2
 ...
```

```
 id | page
----+-----
  1 | 1
  2 | 1
  3 | 1
  4 | 2
  ...
```



The index is cached in RAM if possible, and stored on disk similar to a regular table.

Two RAM lookups, one disk read.
Much better than a table scan.

Index lookup vs. table scan.

GOOD!

```
EXPLAIN SELECT * FROM prospects WHERE id = 5;
QUERY PLAN
-----------------------------------------
 Index Scan using prospects_pkey on prospects
(cost=0.09..4.09 rows=1 width=288)
   Index Cond: (id = 5)
```

GOOD!

```
EXPLAIN SELECT * FROM prospects WHERE rainmaker_id = 5;
QUERY PLAN
-----------------------------------------
 Index Scan using index_prospects_on_rainmaker_id on prospects
(cost=0.09..240.16 rows=179 width=288)
   Index Cond: (rainmaker_id = 5)
```

BAD!

```
EXPLAIN SELECT * FROM prospects WHERE network_id = 5;
QUERY PLAN
-----------------------------------------
 Seq Scan on prospects  (cost=0.00..74054.92 rows=180 width=288)
   Filter: (network_id = 5)
```

Indexes are not free!

- **They take up disk space!**
  Now we're storing an extra copy of both the indexed column and the primary key.
- **They consume RAM!**
  This leaves less memory for caching other things.
- **They require bookkeeping!**
  An index takes cycles to create and keep current.

Indexing a low cardinality field may not be useful.

Cardinality: Number of unique values.

`cardinality(boolean) == 2`

If your index selects nearly half of your records, it will probably result in a full table scan.

```
bool_column | id
------------+-----
false       | 1
false       | 2
false       | 3
false       | 5
false       | 6
false       | 7
false       | 8
false       | 10
false       | 11
...
true        | 4
true        | 9
true        | 12
true        | 13
true        | 14
true        | 15
...
```

Indexing a text field will still be slow-ish for suffix matches.

Fast

```
SELECT * WHERE myfield = 'foo'
SELECT * WHERE myfield LIKE 'foo%'
```

Slow (but better than with no index)

```
SELECT * WHERE myfield LIKE '%foo'
SELECT * WHERE myfield LIKE '%foo%'
```

```
myfield  | id
---------+-----
...
foobar   | 10
kungfoo  | 13
tomfooha | 2
...
```

This works is fine for a small-ish number of rows, depending on how much RAM and how many cores you have.

Once you get to hundreds of thousands of rows, switch to something like ElasticSearch.

Thought Experiments

- Can you index on more than one column?

  ```
  add_index :prospects, [:rainmaker_id, :status]
  ```

  Yes! This is called a composite or multi-column index.

- Assuming you've created the index above, what happens in the following query:

  ```
  SELECT COUNT(id)
  FROM prospects
  WHERE rainmaker_id = 1
  AND status = 2;
  ```

  You have a “covering index” for this query.
  The index includes all columns necessary to answer the query.

Thought Experiments

- You have a comments table with the following fields:
  - id:integer
  - post_id:integer
  - body:string
- Should you index the post_id?
- Should you index the body?

Closing Thoughts

- Learn to interpret the “EXPLAIN” command! http://www.postgresql.org/docs/9.0/static/using-explain.html

- With the right data types, the right indexes, and a reasonable number of columns in a table, the default Postgres settings are good enough for 99% of all cases, even with millions of rows.

- If necessary, an expert DB admin can tweak server level, database level, table level, index level, and connection level settings to optimize performance. The 'fillfactor' setting is just one example.

- None of this is magic, and it's no more difficult than learning about Rails.