40.317 Lecture 6

Dr. Jon Freeman

4 June 2020

Slido Event #5037

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Agenda

- Basics of database optimisation

DB Optimisation: Motivation

Why are we studying this topic?

- Data has become every organisation's most important asset.
- Relational databases ("RDBs") are used by companies in every size & industry.
- RDBs are not cutting edge, but are often all you need.
- Making effective use of an RDB requires some minimal knowledge.

Out of Scope

- Optimising database usage of space as well as time

- General principles of database design

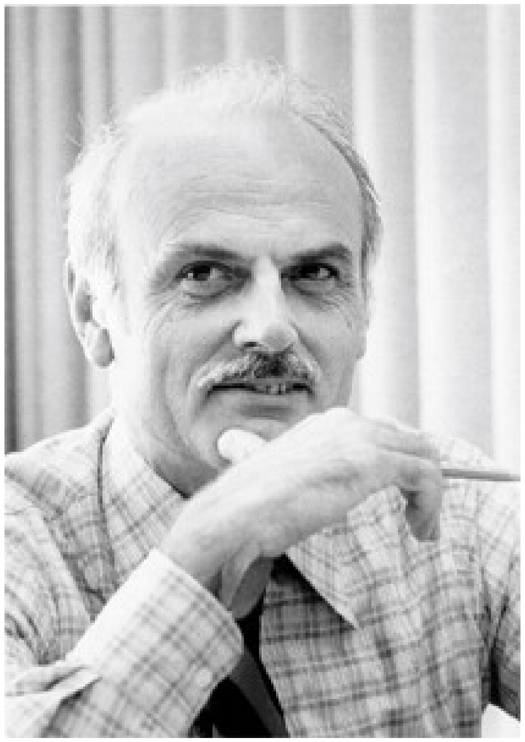
An RDB is a Marvel of Our Age

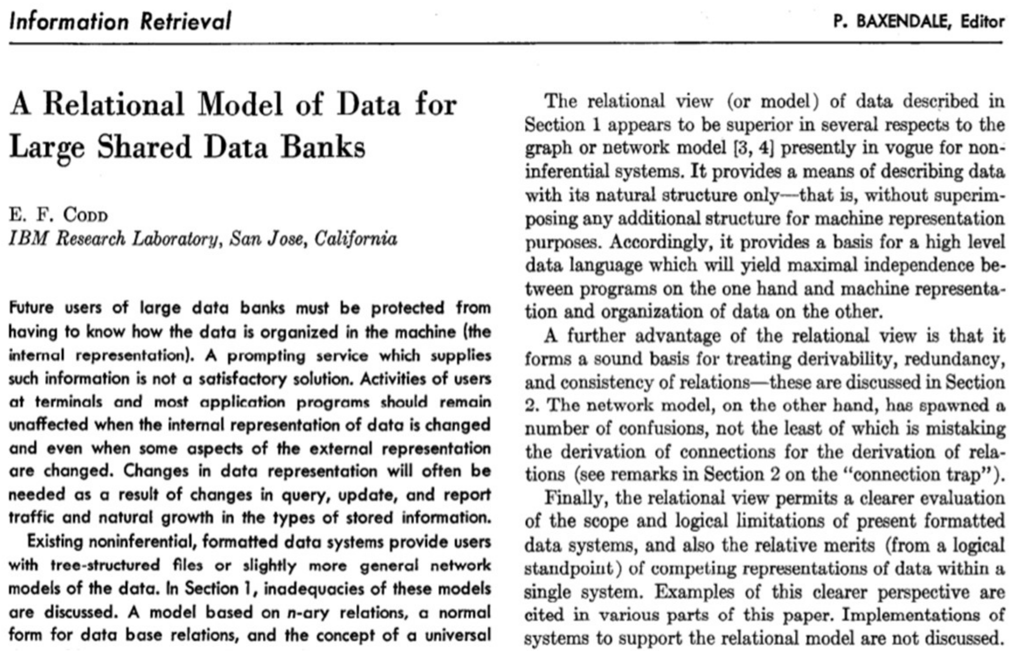
A modern RDB contains or embodies:

- A relational model of data based on relational algebra, a rigorous mathematical framework
- The B-tree data structure, for fast lookups against data in slow storage
- Synchronisation mechanisms to handle competing requests among multiple readers and writers
- Other practical challenges, which must *all* be overcome for an RDB to be fit for purpose

Relational Algebra / Relational Model

Invented by Edgar F. “Ted” Codd at IBM in the 1960s.





Information Retrieval P. BAXENDALE, Editor

A Relational Model of Data for Large Shared Data Banks

E. F. Codd
IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.

Existing noninferential, formatted data systems provide users with tree-structured files or slightly more general network models of the data. In Section 1, inadequacies of these models are discussed. A model based on *n*-ary relations, a normal form for data base relations, and the concept of a universal

The relational view (or model) of data described in Section 1 appears to be superior in several respects to the graph or network model [3, 4] presently in vogue for non-inferential systems. It provides a means of describing data with its natural structure only—that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other.

A further advantage of the relational view is that it forms a sound basis for treating derivability, redundancy, and consistency of relations—these are discussed in Section 2. The network model, on the other hand, has spawned a number of confusions, not the least of which is mistaking the derivation of connections for the derivation of relations (see remarks in Section 2 on the “connection trap”).

Finally, the relational view permits a clearer evaluation of the scope and logical limitations of present formatted data systems, and also the relative merits (from a logical standpoint) of competing representations of data within a single system. Examples of this clearer perspective are cited in various parts of this paper. Implementations of systems to support the relational model are not discussed.

Source: https://en.wikipedia.org/wiki/Edgar_F._Codd

Basics of Relational Algebra

Relational algebra is a formal system for operating on relations, where a relation is a set of tuples. Its possible operations are:

Symbol(s)	Operator(s)
π, σ	Project, Select
ρ	Rename
∪, ∩, -, ÷	Union, Intersection, Difference, Division
←	Assignment
x	Cartesian product
⋈	(Inner) join
⟕, ⟖, ⟗	Left outer join, right outer join, full outer join
⋉, ⋊	Left semijoin, right semijoin

B-Trees

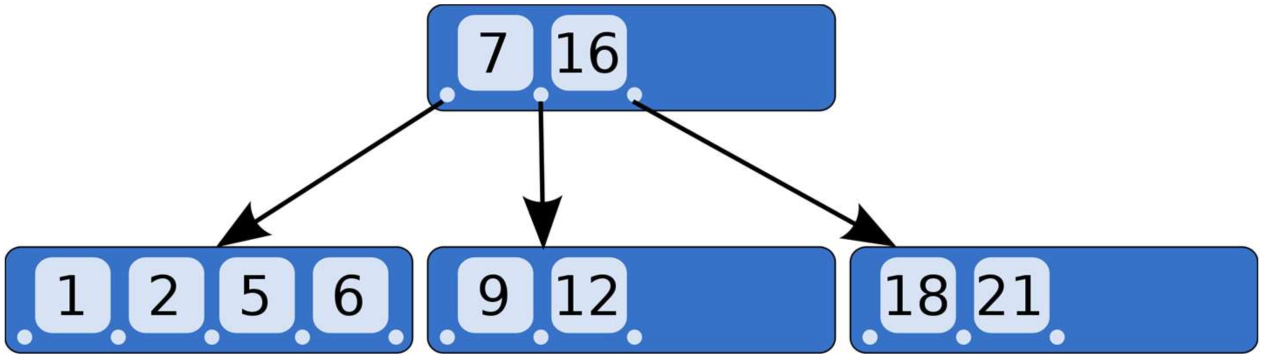
- A self-balancing tree data structure.
  - A generalisation of a *binary* search tree, in that a B-tree node can have more than two children.
- Invented in 1971 by Rudolf Bayer and Ed McCreight at Boeing Research Labs.
  - They never said what, if anything, the “B” stands for ☺

B-Trees, continued

- Maintains sorted data and allows searches, sequential access, insertions, and deletions in $O(\log N)$ time, where N is the total number of records.

- Well suited for storage systems that read and write relatively large blocks of data.
  – Commonly used in file systems as well as databases.

B-Trees, continued



- Each internal (non-leaf) node contains several keys.
- The keys act as separation values which partition the data in the node’s subtrees.
- A node with k keys has k+1 subtrees.

Source: https://en.wikipedia.org/wiki/B-tree

B-Trees, continued

- All leaf nodes must be at the same depth; the tree is re-balanced as needed to ensure this.

- Their primary benefit is speed:
  - Remember that every node is in slow storage, not just the data in the leaves.
  - Having multiple keys in each node minimises the tree’s depth, which in turn minimises the number of slow node accesses.

Synchronisation Mechanisms

In general there are multiple readers, multiple writers, and each of them wants the *illusion* that they have exclusive access to the server, i.e. with at most a minor decrease in performance.

So the authors of an RDB have to tackle some of the same challenges as the authors of an operating system, such as:

Synchronisation Mechanisms

- Deadlock: Two users both attempt to update two tables, T1 and T2, at about the same time.
  - The first user locks T1 and is waiting to lock T2.
  - The second user locks T2 and is waiting to lock T1.
  - Now what?
- Starvation: Multiple users are making a long series of updates to a table, and one of them never gets a turn.

Other Practical Challenges

Here are just a few:

- The data is typically stored on slow disks. How can an insert or update complete quickly in spite of this, i.e. without waiting for every single change to be actually made on the disk itself?

  ⇒ <u>Clean</u> vs <u>dirty</u> pages

Other Challenges, continued

- How can we prevent data corruption when a large, long-running update fails part-way through?

  ⇒ <u>Transactions</u>

- How can we minimise conflicts and delays among the writers, i.e. the users making changes?

  ⇒ <u>Row</u> vs <u>page</u> vs <u>table</u> locking

SQL is a "Leaky Abstraction"

- SQL is "declarative" (as opposed to "imperative"): an SQL query indicates *what* we want, not *how* to obtain it.
  - From an academic perspective, an RDB is an engine for doing relational algebra, and submitting correct queries is enough.
- But inevitably some unwanted behaviour will force us to understand the *how*.
- Joel Spolsky invented the term "leaky abstractions" to refer to this phenomenon.

What Do We Need To Know?

- We realise the *how* behind SQL can be extremely complicated.
- As users, however, we would rather not have to know lots of details.
- What is the minimum we ought to know in order to make effective use of an RDB?

Source: https://www.motorcities.org

Where Our Examples Come From

We will use:

- MySQL as our example RDB.
- The Employees sample database, available from GitHub, because it has some suitably large tables:

```
select table_name, table_rows from
information_schema.TABLES where
table_schema = 'employees' and
table_type = 'BASE TABLE' order by
table_rows desc;
```

Primary Keys

A table's <u>primary key</u> is a set of one or more columns whose values are collectively unique for every row.

- A table can only have one primary key.
- None of the columns in the primary key can contain NULL values.

The columns in a table's primary key are usually obvious and intuitive.

- Let's look in the `employees` DB
- A counterexample from real life

Indexes

An <u>index</u> is a data structure (a B-tree) on disk, associated with a table or a view, which greatly speeds up certain patterns of lookups against it.

There are two types of indexes, <u>clustered</u> and <u>non-clustered</u>.

Clustered Indexes

A clustered index on a table is a special index which reflects how the table’s data is physically arranged on disk.

Two data items close to each other vis-à-vis the index will be close to each other on disk, and vice-versa.

Clustered Indexes, continued

- A table can have at most one clustered index.
- A clustered index does not require additional space, because the table *is* the [leaf level of the] clustered index.
- Changing a clustered index requires rearranging the table’s contents on disk.

Primary Keys vs Clustered Indexes

A primary key is a *constraint*, i.e. a logical entity.

A clustered index is a *data structure*, i.e. a physical entity.

When you create a table with a primary key or unique key, MySQL automatically creates a clustered index named `PRIMARY`.

Non-Clustered Indexes

Determining a table's primary key, and creating a corresponding clustered index, does not ensure adequate performance of all current and future queries against it.

- A non-clustered index is an additional index on any set of columns, to improve the speed of queries on those columns.
  - Presumably such queries occur often.

Non-Clustered Indexes, continued

- Unlike a clustered index, a non-clustered index requires additional space.
  - Reading its contents will therefore take additional time.

- A non-clustered index need not be unique.
  - The “first name + last name” example can help you remember this

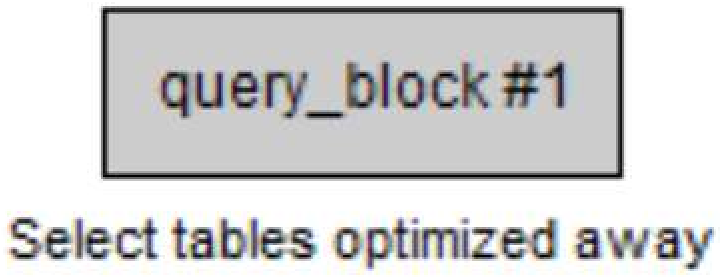
Estimated Query Costs in MySQL

- MySQL has an EXPLAIN command which generates a query's expected cost.
- MySQL Workbench has a visual interface, called Visual Explain, which runs EXPLAIN and then presents a colour-coded summary.
  - Press Ctrl-Alt-x to run Visual Explain on the current query.

Next we present some example Visual Explain outputs, from fast to slow.

Example Queries from Fast to Slow

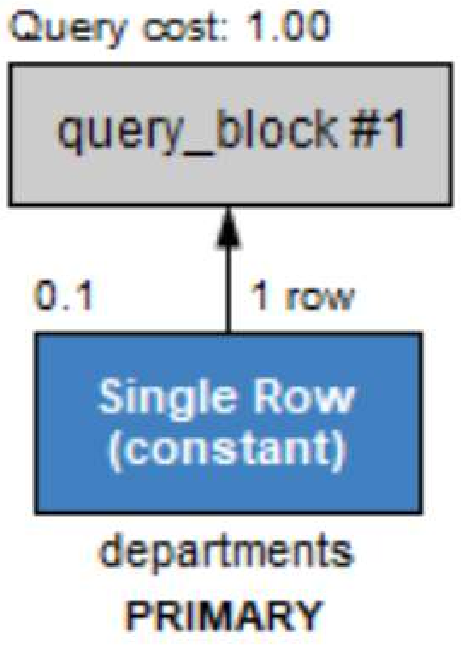
```
select max(from_date) from
salaries where emp_no = 18888;
```



Its cost is so low, Visual Explain does not even assign it a colour!

Example Queries from Fast to Slow

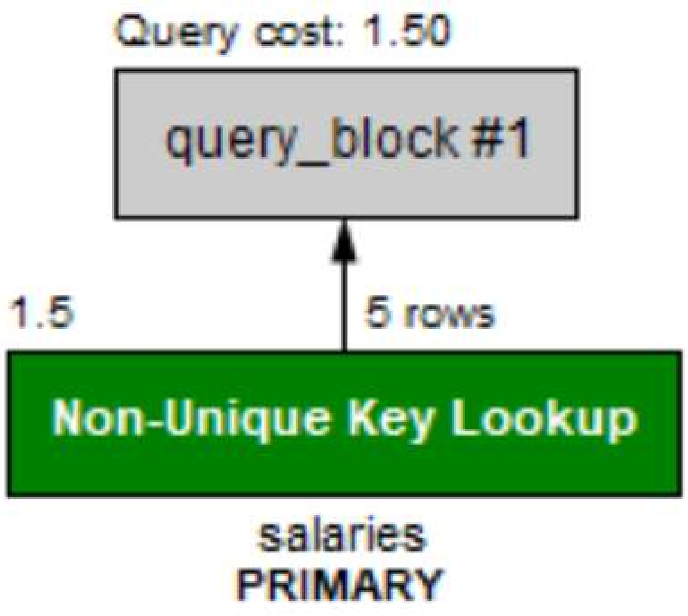
```
select * from departments where
dept_no = 'd008';
```



Blue colour denotes very low cost.

Example Queries from Fast to Slow

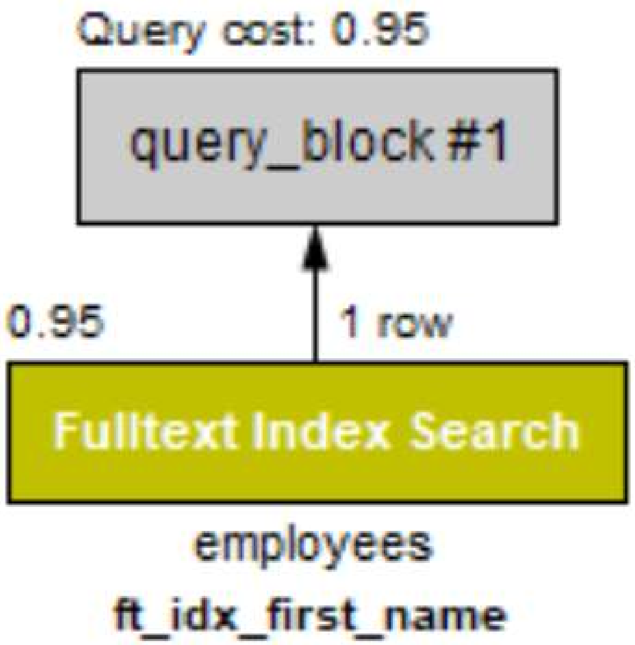
```
select * from salaries where
emp_no = 18888;
```



Green colour denotes “low-medium” cost.

Example Queries from Fast to Slow

```
select * from employees where
match(first_name) against('Mary');
```



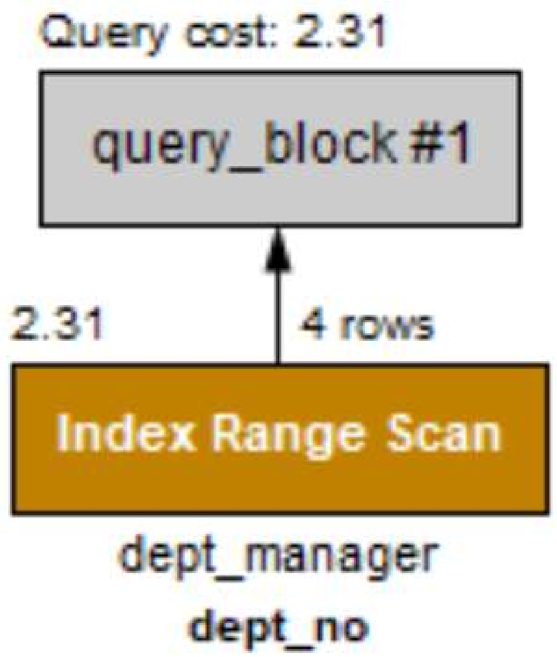
Yellow is for a special FULLTEXT index:

```
create fulltext index ft_fn on employees(first_name);
```

A BETTER WORLD BY DESIGN.

Example Queries from Fast to Slow

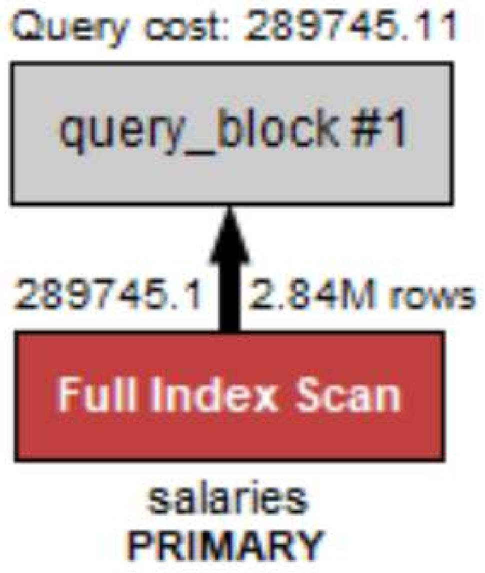
```
select * from dept_manager where
dept_no in ('d005', 'd008');
```



Orange colour denotes medium cost.

Example Queries from Fast to Slow

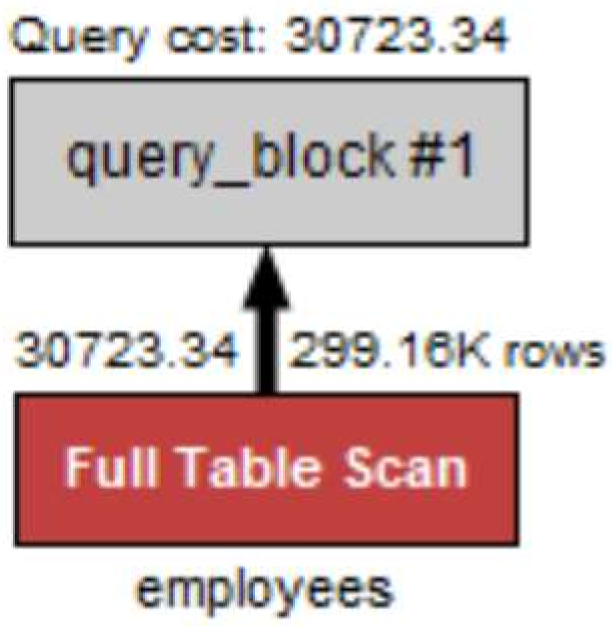
```
select emp_no from salaries where
from_date between '1999-12-01' and
'1999-12-31';
```



Red colour denotes high cost.

Example Queries from Fast to Slow

```
select max(hire_date) from
employees;
```



What is an Index Scan?

An <u>index seek</u> is a search in which only a minimal, relevant portion of an index is accessed.

An <u>index scan</u> is the opposite, i.e. a search in which either much or all of an index is accessed.

- Visual Explain called these "index range scan" and "full index scan" respectively in our examples.

More Index Scan Definitions

An index scan *without row lookups* is a search which involves scanning an entire index, while not reading any table pages.

- `select col2 from T` (there is an index on <col1, col2, col3>)

An index scan *with row lookups* is a search which involves scanning an entire index and also reading some table pages.

- `select col4 from T where col2 = 88` (there is an index on <col1, col2, col3>)

What is a Table Scan?

If a table has no index at all, or no index which can help speed up a particular query, the server will perform that query by conducting a table scan, which is just what it sounds like and can hurt performance, even for moderately sized tables.

Preventing (slow, unwanted) table scans is a basic part of RDB optimisation.

Where did the costs come from?

The Visual Explain outputs included a *numerical* cost estimate for each query. How did MySQL compute them?

How MySQL Estimates Query Costs

First of all, by “cost of a query” we actually mean “cost of a query *plan*.” No, we really mean “cost of *the lowest-cost* query plan.”

The existence of the imperative plan behind each declarative query is what makes SQL a “leaking abstraction,” as these plans are

- not immediately visible;
- of great practical importance; and
- can sometimes be less than ideal.

How MySQL Estimates Query Costs

MySQL selects the query plan with lowest estimated total cost as follows:

1) Assign a cost to each basic operation.
2) Select a small set of plausible plans to consider closely. For each:
    a) Estimate how many basic operations the plan would take.
    b) Sum up the total.
3) Select the plan with lowest total cost.

How MySQL Estimates Query Costs

Interesting: MySQL stores the assumed costs of various operations in (two) tables!

```
select cost_name, default_value from
mysql.server_cost
union
select cost_name, default_value from
mysql.engine_cost
order by default_value desc;
```

More interesting, and somewhat dangerous: we can override these values!

An Optimisation Strategy Takes Shape

In general, index seeks are preferable to index scans, which are preferable to table scans.

In MySQL, a simple yet effective strategy is

- eliminate every red query, and
- minimise the cost of every orange query.

With this in mind, let’s revisit our three red and orange examples.

Applying Our Optimisation Strategy

To eliminate this “red” table scan:

```
select max(hire_date) from
employees;
```

we can

```
create index idx_employees_hd on
employees(hire_date);
```

Applying Our Optimisation Strategy

To eliminate this “red” full index scan:

```
select emp_no from salaries where
from_date between '1999-12-01' and
'1999-12-31';
```

we can

```
create index idx_salaries_fd_en on
salaries(from_date, emp_no);
```

Applying Our Optimisation Strategy

Lastly, let’s re-examine our “orange” example, involving an index range scan:

```
select * from dept_manager where
dept_no in ('d005', 'd008');
```

Is there anything we could, or should, do?

More About Index Range Scans

When an index involves *multiple* columns, we must choose their ordering with care.

Our objective should be:

- Order the columns in an index to minimize the scanned index range which our typical queries will generate.

More specifically:

- Index for equality first, then for ranges.

Actual vs. Estimated Query Costs

Of course MySQL can show you a query's actual cost as well as its estimated cost:

- Click the Query Stats tab in MySQL Workbench, or
- `set profiling=1; <your query>; show profile;`

These outputs are more accurate, though less friendly, than Visual Explain's.

Statistics

Visual Explain's outputs are estimates, so we need them to be accurate estimates.

What affects their accuracy more than anything else is the server's <u>statistics</u>:

- Think "meta-data and histograms".
- They are internal, and cached.
- They become outdated over time, but can be re-generated as needed.

Our example DB is not well designed

Principles of DB design are out of scope, but for the employees DB I must observe:

- `dept_emp` and `dept_manager` should be combined into a single table
- `employees.hire_date` has no corresponding end date, so only *current* employees can be recorded
- Actual titles should be moved to a separate table with a numeric `title_no`
- `dept_no` can, hence should, be numeric

More employees DB design flaws

- This design only captures *department* managers, but surely there are other types of managers.
- (Hence) this design does not capture all reporting relationships; why not do so?
- In general this design gives too much importance to departments, i.e. assumes too much about company structure.
  - What about teams within departments?
  - What about reporting lines across departments?

Takeaways

(Recurring) table scans should be avoided (for larger tables), and always can be avoided.

Every table we create should have at least one index, namely the (one and only) clustered index on its primary key columns.

- MySQL's default InnoDB storage engine requires a primary key; if you don't specify one, it creates a hidden one for you.

Takeaways, continued

We can create multiple non-clustered indexes on the same table. Such indexes need not be unique. The main tradeoffs:

- Each one takes up space.
- Modifying them all after an insert, update, or delete takes that much longer.

Takeaways, continued

Having lots of indexes on a table does not guarantee an absence of performance problems. The key is our *usage patterns*, i.e. our typical queries, and whether the indexes we have in place cover them all.

- N.B. MySQL will only ever use one index per table per query, so you will need one index for each distinct usage pattern.

Takeaways, continued

Regardless of our usage patterns or how they evolve, over time

- indexes become fragmented, and
- statistics become outdated,

so from time to time we must de-fragment indexes and/or update statistics. In MySQL:

- `optimize table` to rebuild a table's indexes and update its statistics
- `analyze table` to only update stats

Takeaways, continued

Our usage patterns change over time as well. With all these ongoing changes, we must monitor our server's query plans and overall health.

The authors of the queries should assist with this monitoring, because they are in the best position to understand and fix their queries and the scripts which invoke them.

Recommended Reading

- MySQL Tutorial
- SQL Performance Explained by Markus Winand (2012); free web edition here
- MySQL Optimisation Overview, https://dev.mysql.com/doc/refman/8.0/en/optimize-overview.html

Thank you

A BETTER WORLD BY DESIGN.

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN