Lecture 12

Database Optimization

Indexes
Non-natural keys
Denormalization

Lab: Stuck waiting for grading of Task X?

“We can not continue with Task X+1 before Task X is accepted, and the TA is taking too long!!”

- TAs are also people with busy schedules and a lot of work.
- No need to wait until Task X is accepted to start working on Task X+1
- Any changes required for Task X must be applied to Task X+1 too, but hopefully those will be minimal

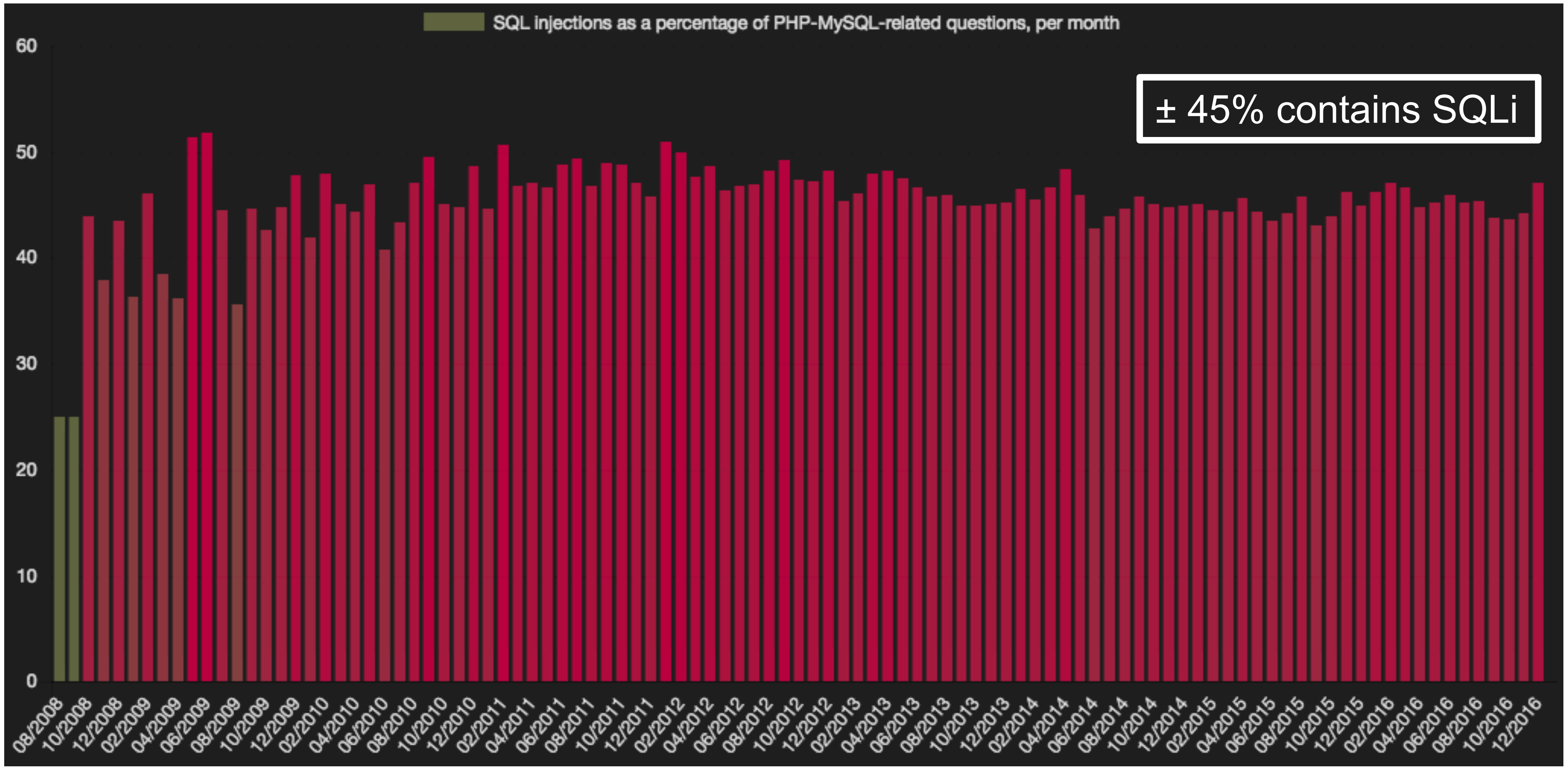
Task 5: procedure

- Show your finished Task 5 before or in the last lab session
  - Prepare your demo, make it go smooth
  - Anticipate that you need to make changes, don’t wait until last minute
  - Only submit Task 5 to Fire after TA’s approval
    - Mention “seen by <TA’s name>” in submission note
    - Submit directly after TA approval
- Demo deadline: Friday 15 December 2016
- Fire deadline: Monday 17 December 2016

Why use Views from JDBC

- Security
  - Separate permissions on VIEW (e.g. read-only, potentially enforced with trigger)
- Hide data
  - Tables may contain sensitive data that not everyone needs access to (e.g. credit card numbers)
- Hide complexity
  - Queries may be complicated with joins, unions, subqueries, etc. This complexity can be hidden with a view
- Support legacy code
  - Refactoring a table may break a lot of code. A view can preserve the original look of a table while the latter changes in the background.

Why you should not blindly copy/paste from Stackoverflow



Source: https://laurent22.github.io/so-injections/

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil“

- Donald Knuth, 1974

This does <u>not</u> imply: do not optimize,
But instead: focus on functionality first, and then optimize

Quiz!

How costly is this operation (naive solution)?

course	*per*	*weekday*	*hour*	*room*
TDA356	2	VR	Monday	13:15
TDA356	2	VR	Thursday	08:00
TDA356	4	HB1	Tuesday	08:00
TDA356	4	HB1	Friday	13:15
TIN090	1	HC1	Wednesday	08:00
TIN090	1	HA3	Thursday	13:15

$\left.\right\} n$

```
SELECT *
FROM   Lectures
WHERE  course = 'TDA357'
       AND period = 3;
```

Go through all *n* rows, compare with the values for course and period = 2*n* comparisons

Quiz!

Can you think of a way to make it faster?

```
SELECT *
FROM   Lectures
WHERE  course = 'TDA357'
       AND period = 3;
```

If rows were stored sorted according to the values course and period, we could get all rows with the given values faster (O(log n) for tree structure).

Storing rows sorted is expensive, but we can use an *index* that given values of these attributes points out all sought rows (an index could be a hash map, giving O(1) complexity to lookups).

Index

- When relations are large, scanning all rows to find matching tuples becomes very expensive.
- An *index* on an attribute A of a relation is a data structure that makes it efficient to find those tuples that have a fixed value for attribute A.
  - Example: a hash table gives amortized O(1) lookups.

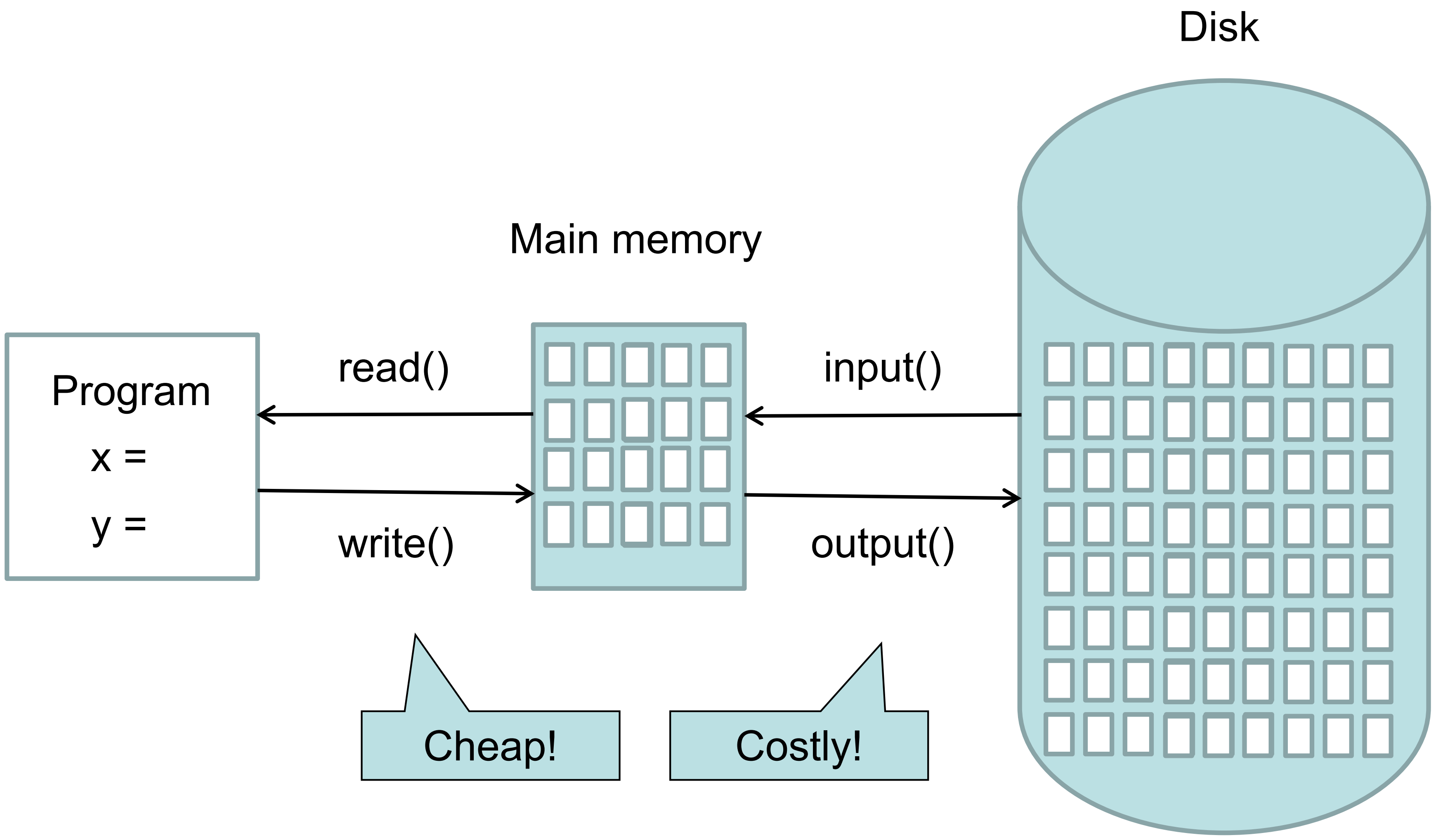
Quiz!

Asymptotic complexity (O(x) notation) is misleading here. Why?

The asymptotic complexity works for data structures in main memory. But when working with stored persistent data, the running time of the data structure, once in main memory, is negligible compared to the time it takes to read data from disk. What really matters to get fast lookups in a database is to minimize the number of disk blocks accessed (could use asymptotic complexity over disk block accessing though).

Indexes help here too though. If a relation is stored over a number of disk blocks, knowing in which of these to look is helpful.

Disk and main memory



Typical costs

- Some (over-simplified) typical costs of disk accessing for database operations on a relation stored over n blocks:
  - Query the full relation: n (disk operations)
  - Query with the help of index: k, where k is the number of blocks pointed to (1 for key).
  - Access index: 1
  - Insert new value: 2 (one read, one write)
  - Update index: 2 (one read, one write)

Example:

```
SELECT *
FROM   Lectures
WHERE  course = ’TDA357’
       AND period = 3;
```

Assume Lectures is stored in *n* disk blocks. With no index to help the lookup, we must look at all rows, which means looking in all *n* disk blocks for a total cost of *n*.

With an index, we find that there are 2 rows with the correct values for the course and period attributes. These are stored in two different blocks, so the total cost is 3 (2 blocks + reading index).

Quiz!

How costly is this operation?

```
SELECT *
FROM   Lectures, Courses
WHERE  course = code;
```

Lectures: n disk blocks

Courses: m disk blocks

No index:
Go through all *n* blocks in Lectures, compare the value for course from each row with the values for code in all rows of Courses, stored in all *m* blocks. The total cost is thus ***n * m*** accessed disk blocks.

Index on code in Courses:
Go through all *n* blocks in Lectures, compare the value for course from each row with the index. Since course is a key, each value will exist at most once, so the cost is ***2 * n + 1*** accessed disk blocks (1 for fetching the index once).

CREATE INDEX

- Most DBMS support the statement

```
CREATE INDEX index name
  ON table (attributes);
```

  – Example:

```
CREATE INDEX courseIndex
 ON Courses (code);
```

  – Statement not in the SQL standard, but most DBMS support it anyway.
  – Primary keys are given indexes implicitly (by the SQL standard).
  – In PostgreSQL, use `\di` to list indexes

Important properties

- Indexes are separate data stored by itself.
    - Can be created
        - ✓ on newly created relations
        - ✓ on existing relations
            - will take a long time on large relations.
    - Can be dropped without deleting any table data.
- SQL statements do not have to be changed
    - a DBMS automatically uses any indexes.

Quiz!

Why don’t we have indexes on all (combinations of) attributes for faster lookups?

- Indexes require disk space.
- Modifications of tables are more expensive.
  - Need to update both table and index.
- Not always useful
  - The table is very small.
  - We don’t perform lookups over it (Note: lookups ≠ queries).
- Using an index costs extra disk block accesses.

Real world

- The examples given here are very simplified! In reality, many more factors matter:
  - Data layout on disk, storage schemes
  - Size of disk blocks
  - Size of main memory
  - Disk latency, bus speed, …
- Indexes can be arbitrarily large!
  - Not uncommon for index to be larger than the data set.
  - Different index schemes also matter.

Summary – indexes

- Indexes make certain lookups and joins more efficient.
  - Disk block access matters.
  - Multi-attribute indexes
- **`CREATE INDEX`**
- Usage analysis
  - What are the expected operations?
  - How much do they cost?

    Σ(cost of operation)x(proportion of operations of that kind)

Natural keys

- A *natural key* is a key consisting of attributes in the domain model.
- In some cases, no suitable natural key exists.
  - No suitably unique natural candidate key.
  - Natural candidate key ”too large”.
  - Natural candidate key ”not stable”.
  - …

Artificial key

- Extra attribute added to a table with the purpose of being the key.
  - Does not exist in ”reality”
  - Can be verified for correctness
  - Can be distinguished from artificial keys on other tables in database.
- Examples:
  - Personal numbers, car registration numbers, course codes, etc.

Surrogate key

- System-generated key to replace the actual key behind the covers.
  - AUTO_INCREMENT, SEQUENCE, IDENTITY, …
  - Totally unrelated to domain.
  - NOT exposed to user modification – database consistency would be at great risk!
    - Remember: From the database perspective, application programmers count as users!
  - Example: post/comment IDs managed by Wordpress.

Exposed locators

- Unholy mix of artificial and surrogate keys:
  - System-generated, non-verifiable value with no relation to data model (like a surrogate key).
  - … but exposed to user (like an artificial key).

”[Exposed locators] are handy for lazy, non-RDBMS programmers who do not want to research or think! This is the worst way to program in SQL.”
Joe Celko, SQL programming guru

http://www.informationweek.com/software/business-intelligence/celko-on-sql-natural-artificial-and-surr/201806814

BEWARE!

- In parts of industry, there is an exaggerated belief in using surrogates, or even exposed locators.

> ”In the real world, outside of school, it is considered insanity to have more than an integer as key.”
>
> *Old student*

- Don’t believe it! There is no one-size-fits-all solution to picking keys. Think for yourselves! You are better than them!

Advantages

- Non-natural keys can be more compact.
  - Smaller references, smaller indexes.
  - Faster comparisons, faster joins.

- Non-natural keys are immutable.
  - Not tied to data in domain, so changes of the data will not cause key to change.
    - (Recall: Oracle does not support ON UPDATE CASCADE)
  - Applications never lose their reference to a particular row in the database.
- …

Disadvantages

- Non-natural keys may degrade performance.
  - An extra key on a table requires an extra index to handle external lookups on the natural key
    - extra disk space to store index
    - modifications become more costly
  - Reference to non-natural key means external lookups on the natural key in referencing table requires one or more extra joins.
- Non-natural keys may make maintenance harder.
  - Harder to spot errors, in keys and in references.
- …

Quiz!

Find all lectures for course TDA357 in period 3. How costly is this operation?

```
Courses(code, …)
GivenCourses(course, period, …)
   course -> Courses.code
Lectures(course, period, weekday, …)
    (course, period) ->
      GivenCourses.(course, period)
```

Indexes for primary keys

```
SELECT *
FROM   Lectures
WHERE  course = 'TDA357'
       AND period = 3;
```

Costs ***k* + *1*** where

- **k** is the number of blocks holding rows matching the values
- **1** for reading index

Quiz!

Find all lectures for course TDA357 in period 3.
How costly is this operation?

```
Courses(cid, code, …)
GivenCourses(gcid, course, period, …)
   course -> Courses.cid
Lectures(lid, gcourse, weekday, …)
   gcourse -> GivenCourses.gcid
```

Indexes for primary and natural keys

```
SELECT *
FROM   Lectures, GivenCourses,
       Courses
WHERE  gcourse = gcid
  AND  course  = cid
  AND  code    = 'TDA357'
  AND  period  = 3;
```

Costs ***k + m + 1 + 3*** where

- **k** is the number of blocks holding matching lectures
- **m** is the number of blocks holding matching given courses
- **1** is the block holding the matching code (natural key so only one)
- **3** for reading three separate indexes

Words of (my) advice

1. Use natural keys.
2. If none available, find or create an artificial key.
   - For (strong and weak) entities only: all tables representing relationships will have natural (composite) keys.
   - Also do this if natural key not suitably immutable.
3. If, and **ONLY** if, you notice a performance problem, surrogate keys *might* help.
   - Remember to mark natural keys unique.
   - Remember to create index for natural key lookups, if needed.
   - Use views to hide the surrogate keys from users. Avoid exposed locators.
   - Never include surrogates in e.g. E-R diagram – they are an implementation detail.

Denormalization

- ”Re-compose” decomposed tables or attributes, to avoid joining.
  - Can think of this as pre-computing joins
  - Trade-off: query speed vs. redundancy
  - Are updates frequent?
  - ”NULLs approach” for sub-entities and many-to-at-most-one is a special case – both composed tables have the same key, so *less* data will be stored.

Summary – optimization

- Indexes
  - (often) speed up queries and joins
  - make modifications more costly
- Natural keys, artificial keys, surrogate keys
  - Avoid exposed locators!
  - Know when to use what.
- Denormalization
  - Can be a worthwhile trade-off.