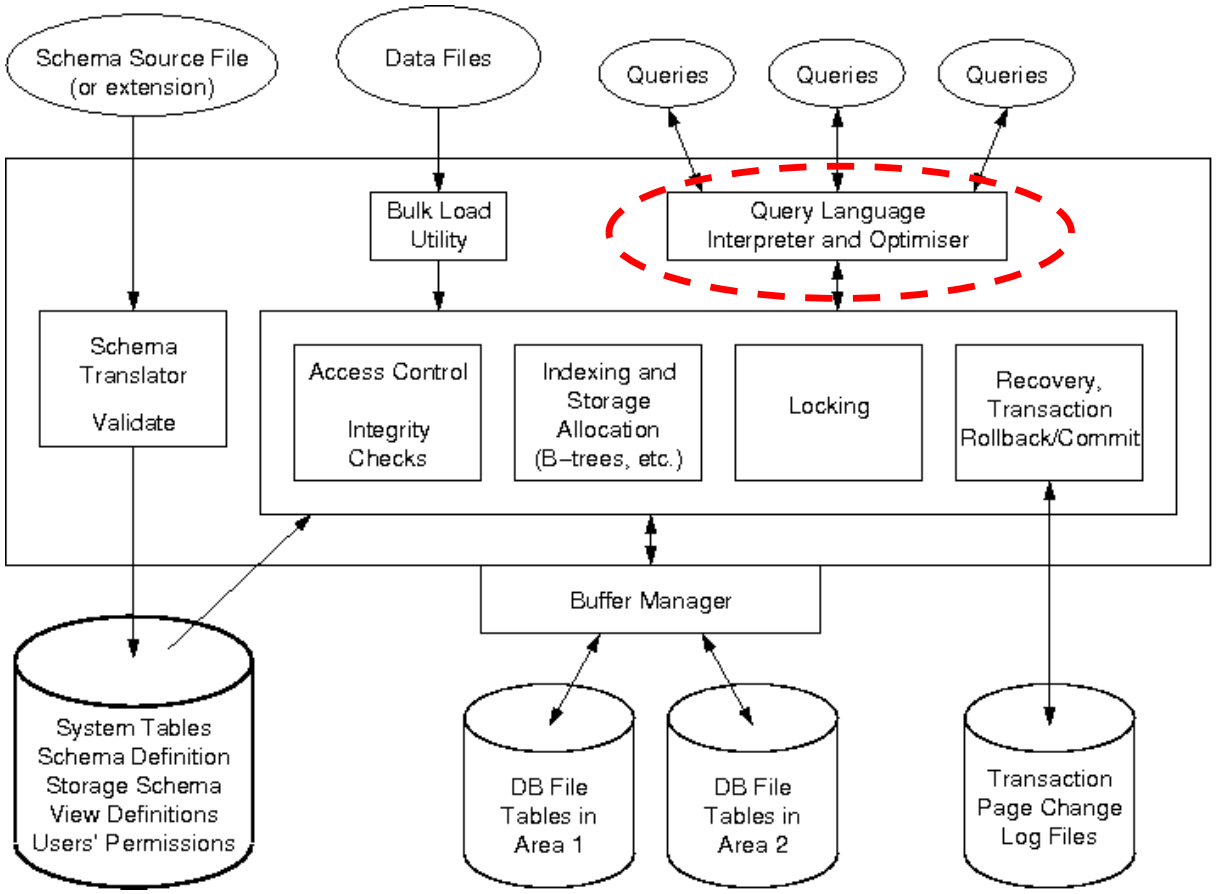
Techniques for Large-scale Data

2019-2020, GK-2

Query processing and optimisation; database design

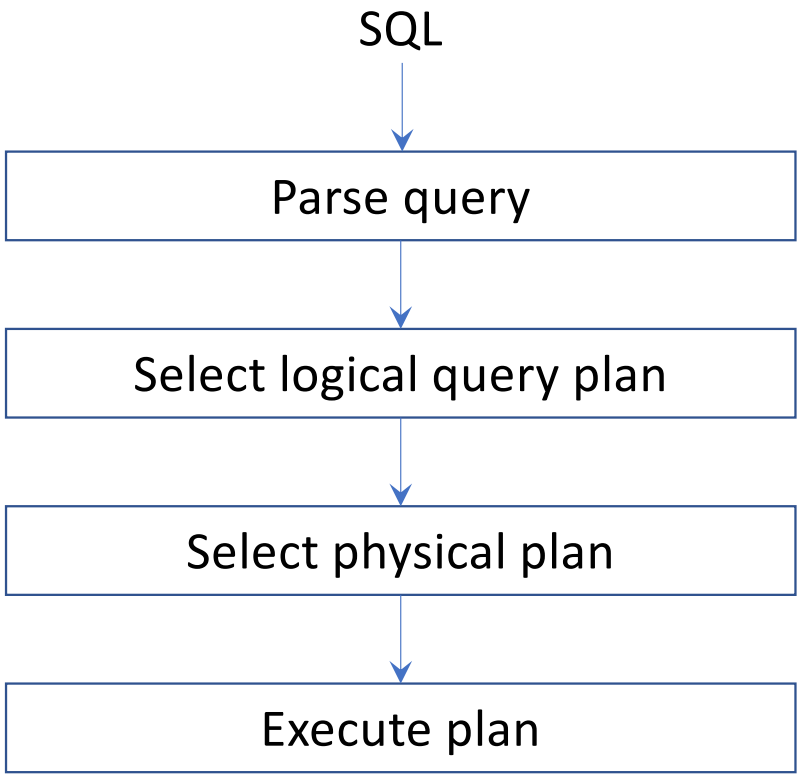
Database system architecture



Declarative query languages

- SQL is a ***declarative*** query language
- Users describe what they want, not how to compute the result
- The database management system decides how to compute the result

Query compilation



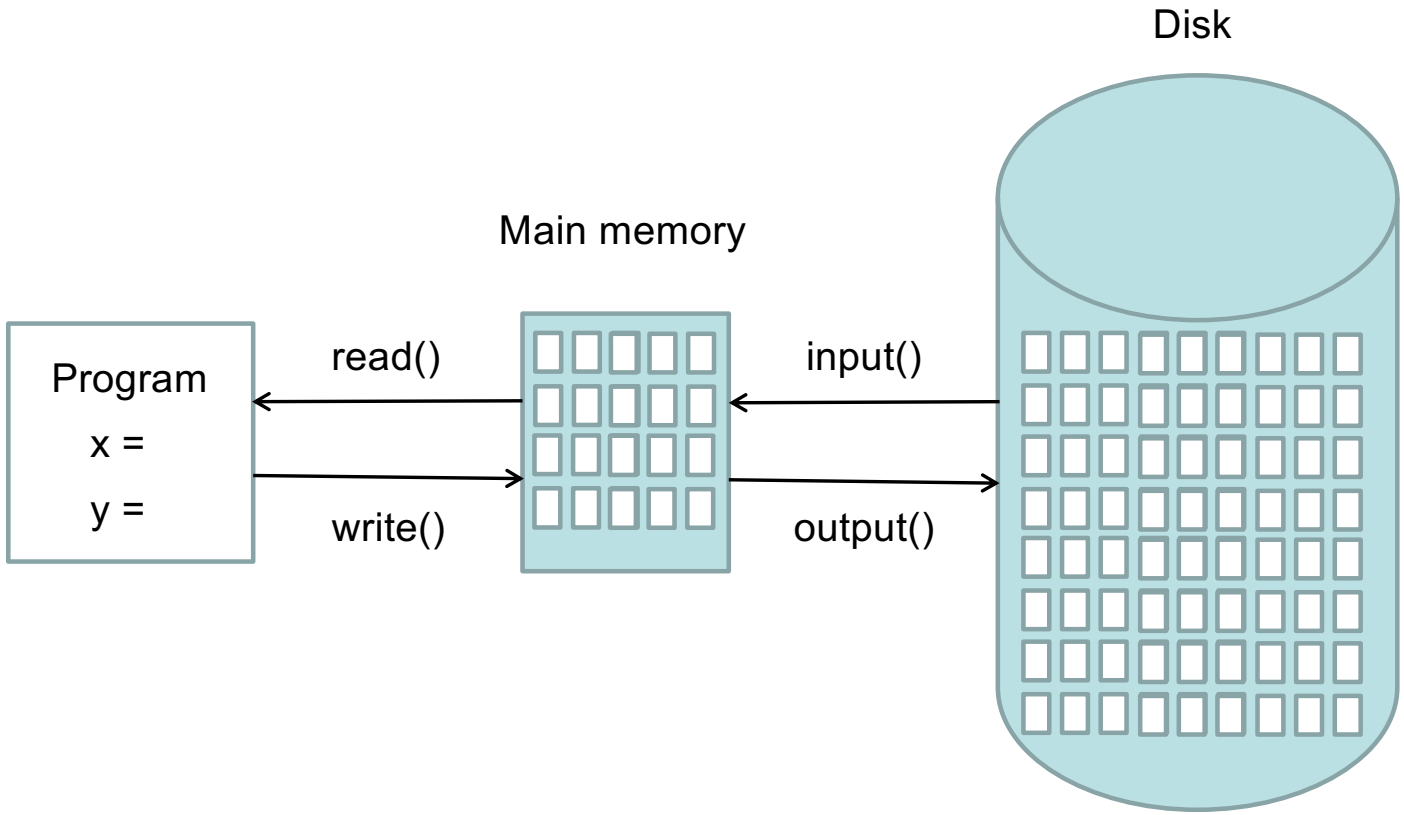
Query expression tree

Logical query plan tree

Physical query plan tree

Physical query plan operators

Block buffer



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.

Scanning tables

Read the entire contents of a relation (the most basic thing to do)

- A *table-scan* involves getting all blocks that contain tuples of the relation, one-by-one.

Read only those tuples that satisfy a predicate

- An *index-scan* involves using an index to get blocks that contain tuples that satisfy the predicate

Table-scan and index-scan are physical query operators

Question 1 (GK-2)

Performing an index-scan is always faster than performing a table-scan.

A. True
B. False

Join algorithms

- Nested-Loop Join
  - For each tuple in the outer relation, scan the inner relation
- Block Nested-Loop Join
  - Each block in the inner relation is paired with each block in the outer relation
  - Their tuples are compared
- Indexed Nested-Loop Join
- Merge Join
- Hash Join
  - Partition tuples of each relation into sets with same hash value on the join attributes
- … and others

Sorting while scanning a table (sort-scan)

- If there exists a B-tree index on the sort attribute(s), scan the index to find tuples in the required order.

- If relation can fit into main memory, use table-scan or index-scan to get all tuples into main memory, then use a main-memory sorting algorithm.

- If relation is too large to fit into main memory, use a multiway merge-sort algorithm

Two-phase multiway merge-sort

Phase 1:

- Repeatedly fill the *M* buffers with new tuples from the relation
- Sort them using a main memory sorting algorithm
- Write out each sorted sublist to disk

Phase 2:

- Merge the sorted sublists, with one input block for each of the sorted sublists (maximum *M-1*) and one output block

Two-pass algorithms based on sorting

The two-phase multiway merge-sort algorithm can be adapted to perform other operations, including:

- Duplicate elimination
- Grouping and aggregation
- Set union, intersection and difference
- Join algorithms

As an alternative to sorting, there are corresponding two-pass algorithms based on hashing.

Query optimisation

Relational operators (1)

- Selection
  - Choose rows from a relation
  - State condition that rows must satisfy

$$\sigma_{condition}(T)$$

Examples:

$\sigma_{seats>100}(Rooms)$

$\sigma_{(code=”TDA143” \text{ AND } day=”Friday”)}(Lectures)$

Relational operators (2)

- Projection
  - Choose columns from a relation
  - State which columns (attributes)

Examples:

$\pi_{A}(T)$

$\pi_{code}(Courses)$

$\pi_{name,seats}(Rooms)$

Relational operators (3)

$R_1 \times R_2$

- Cartesian product
- Combine each row of R_1 with each row of R_2

$R_1 \bowtie_{condition} R_2$

- join operator
- Combine row of R_1 with each row of R_2 if the condition is true

$$R_1 \bowtie_{condition} R_2 = \sigma_{condition}(R_1 \times R_2)$$

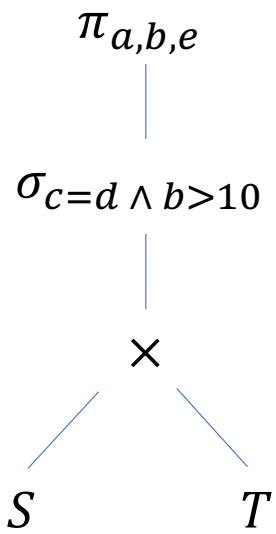
SQL and Relational Algebra

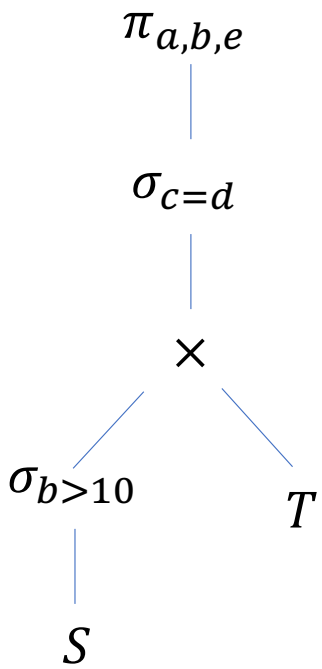
- SQL is based on relational algebra.
  - Operations over relations
- SELECT-FROM-WHERE-GROUPBY-HAVING-ORDERBY
- Operations for:
  - Selection of rows (σ)
  - Projection of columns (π)
  - Combining tables
    - Cartesian product (x)
    - Join, natural join, outer join ($\bowtie_C$, $\bowtie$, $\overset{\circ}{\bowtie}$)
  - Grouping and aggregation
    - Grouping (γ)
    - SUM, AVG, MIN, MAX, COUNT
  - Set operations
    - Union (∪)
    - Intersect (∩)
    - Set difference (\)
  - Miscellaneous
    - Renaming (ρ)
    - Duplicate elimination (δ)
    - Sorting (τ)
- Subqueries

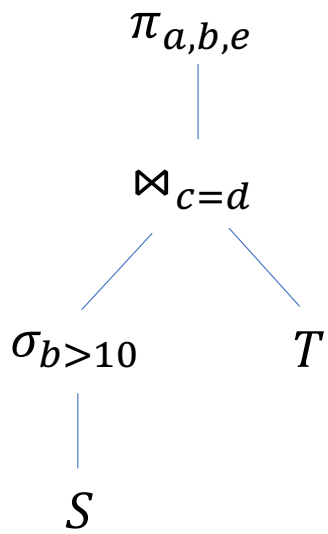
Logical query plans

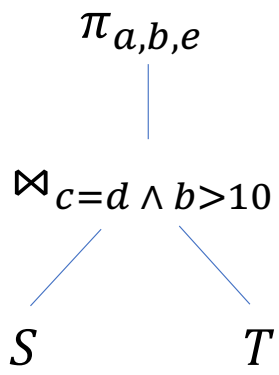
Suppose we have relations S(a,b,c) and T(d,e).

```
SELECT a,b,e
FROM   S,T
WHERE  c=d AND b>10
```









Equivalence rules (1 of 6)

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}\left(\sigma_{\theta_2}(E)\right)$$

2. Selection operations are commutative.

$$\sigma_{\theta_1}\left(\sigma_{\theta_2}(E)\right) = \sigma_{\theta_2}\left(\sigma_{\theta_1}(E)\right)$$

3. Only the last in a sequence of projection operations is needed, the others can be omitted.

$$\pi_{t_1}\left(\pi_{t_2}\left(\ldots\left(\pi_{t_n}(E)\right)\ \ldots\right)\right) = \pi_{t_1}(E)$$

Equivalence rules (2 of 6)

4. Selections can be combined with Cartesian products and theta joins.

   a. $\sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2$

   b. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

5. Theta-join operations (and natural joins) are commutative.

$$E_1 \bowtie_{\theta} E_2 = E_2 \bowtie_{\theta} E_1$$

Equivalence rules (3 of 6)

6. a) Natural join operations are associative:

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

b) If θ_2 involves only attributes from E_2 and E_3 then theta joins are associative as follows:

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

Equivalence rules (4 of 6)

7. The selection operation distributes over the theta join operation under the following two conditions:

   (a) When all the attributes in θ_1 involve only the attributes of one of the expressions (E_1) being joined.

$$\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_{\theta_2} E_2$$

   (b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_{\theta} (\sigma_{\theta_2}(E_2))$$

Equivalence rules (5 of 6)

8. The projections operation distributes over the theta join operation as follows:

   Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively.

   (a) if θ involves only attributes from $L_1 \cup L_2$:

$$\pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \left(\pi_{L_1}(E_1)\right) \bowtie_\theta \left(\pi_{L_2}(E_2)\right)$$

   (b) Consider a join $E_1 \bowtie_\theta E_2$.

   - Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and
   - Let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$.

$$\pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \pi_{L_1 \cup L_2}\left(\left(\pi_{L_1 \cup L_3}(E_1)\right) \bowtie_\theta \left(\pi_{L_2 \cup L_4}(E_2)\right)\right)$$

Equivalence rules (6 of 6)

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over $\cup$, $\cap$ and $-$.

$$\sigma_\theta (E_1 - E_2) = \sigma_\theta (E_1) - \sigma_\theta(E_2)$$

and similarly for $\cup$ and $\cap$ in place of $-$

Also: $\sigma_\theta (E_1 - E_2) = \sigma_\theta(E_1) - E_2$

and similarly for $\cap$ in place of $-$, but not for $\cup$

12. The projection operation distributes over union

$$\pi_L (E_1 \cup E_2) = (\pi_L (E_1)) \cup (\pi_L (E_2))$$

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1 \ ?$$

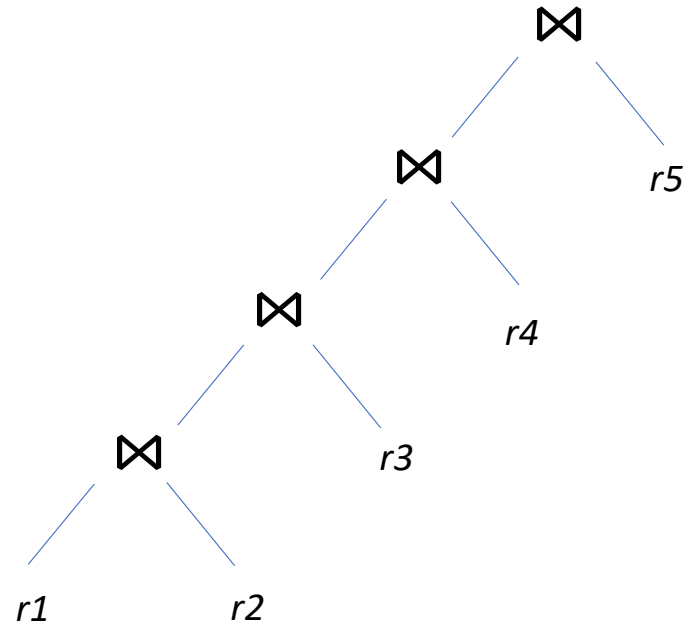
- In a logical query plan, joins are commutative
- In a physical query plan, the relations on the left (the “outer” relation) and right (the “inner” relation) play different roles:
  - Scan through the tuples in the relation on the left
  - Find matching tuples in the relation on the right

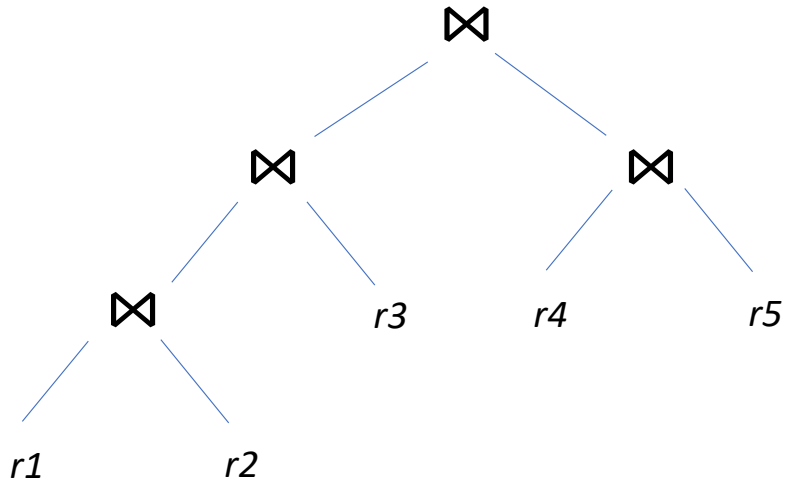
Join ordering

- Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ldots r_n$

- There are $(2(n-1))!/(n-1)!$ different join orderings.
  - With $n = 7$, there are 665280
  - With $n = 10$, there are >176 billion
- Left and right join arguments play different roles; not symmetric
- Left-deep join trees
  - Fewer trees to be considered
  - Fit well with common join algorithms

Left-deep join tree





Non-left-deep join tree

Question 2 (GK-2)

Suppose a query joins three relations.

How many left-deep join trees are there for this query?

A. 1
B. 3
C. 6
D. 9
E. 12

Query optimisation

Logical query plan generation

- Which of the algebraically equivalent forms of a query lead to the most efficient algorithm for answering the query?

Physical query plan generation

- For each operation (e.g. each join step) of the selected form, what algorithm should we select to implement that operation?
- How should data be passed from operation to another, e.g. in a pipelined fashion, in main-memory buffers, or via the disk?

Choices depend on size of relations, approximate frequency of different values for an attribute, availability of indexes, layout of data on disk.

Optimisation in PRTV (mid 1970s)

1. Perform selections as early as possible
2. Perform projections as late as possible
3. Combine selections and projections applying to the same relation
4. Combine selections and projections with union and difference to use a “one-pass” method where possible
5. Move projections in and evaluate the costs of moving them down to various depths, allowing for the improvements to joins resulting from working with smaller numbers of smaller sorted tuples
6. Reconsider forming compound selections
7. Spot common subexpressions, evaluate once and store the value

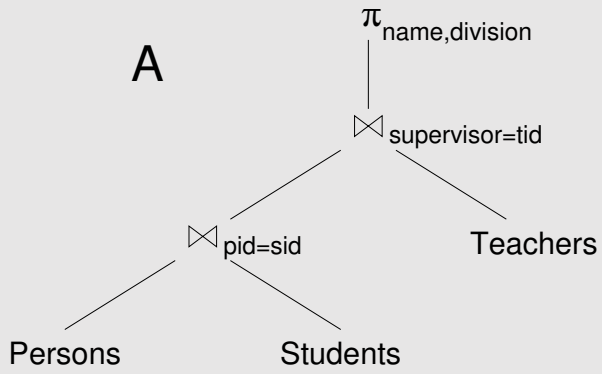
Persons(pid, name) Teachers(tid, division) Students(sid, supervisor)

Question 3 (GK-3)

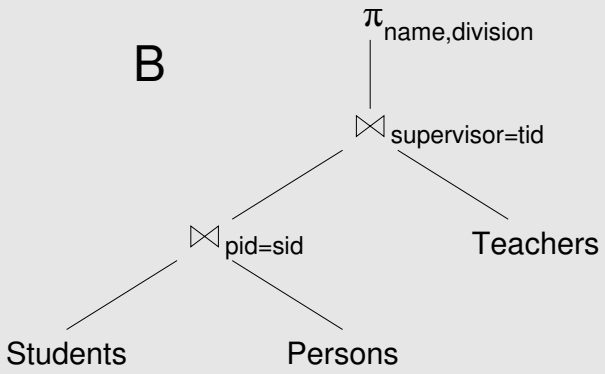
Assume that there are only indexes on the key attributes, and that relations Students and Teachers have approximately the same number of rows.

Which of these query plans is slowest?

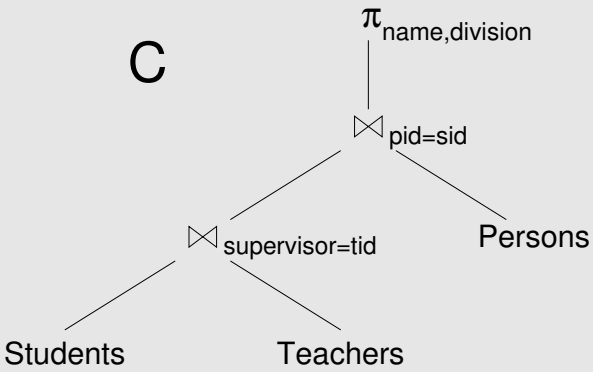
A



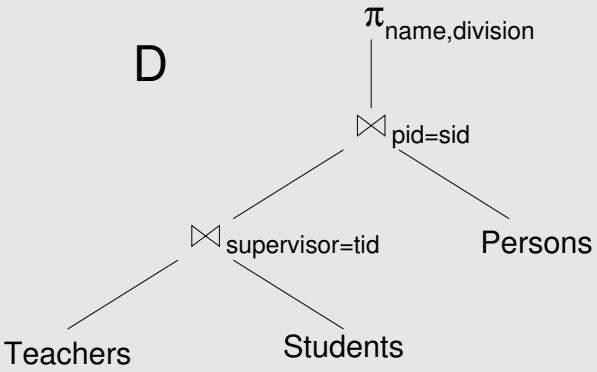
B



C



D



Lectures(course, period, weekday, hour, room)

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

n

```
SELECT *
FROM   Lectures
WHERE  course = 'TDA356'
       AND period = 2;
```

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

Lectures(course, period, weekday, hour, room)

How to make it faster?

```
SELECT *
FROM   Lectures
WHERE  course = 'TDA356'
       AND period = 2;
```

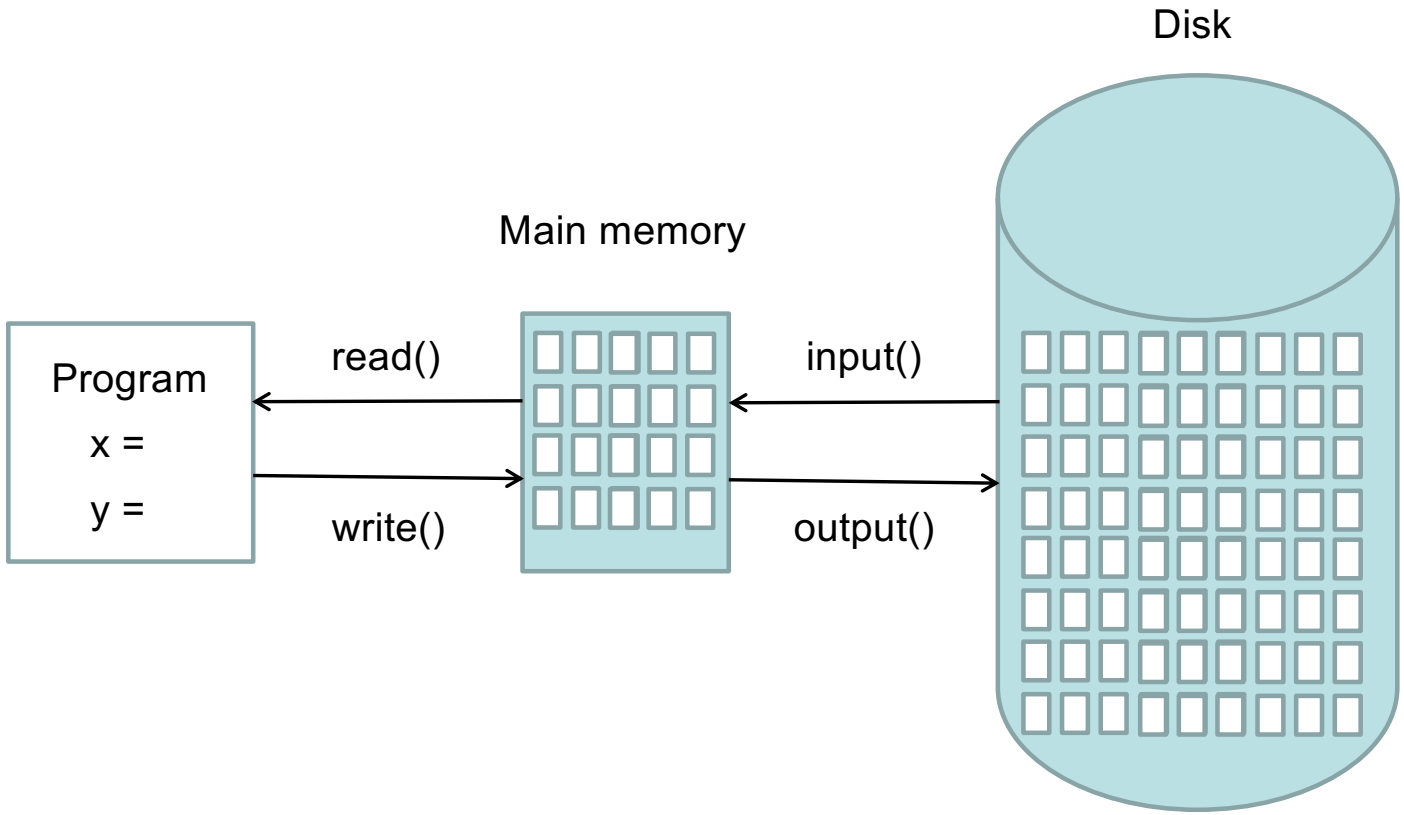
If rows were stored sorted according to the values course and period, we could get all rows with the given values faster (O(log n) for tree structure).

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).

Typical costs

- Some 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)

Block buffer



Lectures(course, period, weekday, room)

Example:

```
SELECT *
FROM   Lectures
WHERE  course = 'TDA356'
       AND period = 2;
```

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).

Lectures(course, period, weekday, hour, room)

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).

“2*n + 1” is a rough indication of query cost

Estimating query execution times is difficult for various reasons, e.g.

- How are data distributed over disk blocks?
- What optimisations will be done?
- Is the database “warm” or “cold”?

If we assume that:

- each disk block that contains part of the Lectures relation contains one tuple,
- each request to find a matching row in the Courses relation requires one block to be brought in from disk, and
- the entire index fits into a single block which must be input() once, and
- the query plan produced has a join with the Lectures relation on he left

Then the cost will be 2*n + 1, but …

(New: 2020-05-31)

Cost can be greater than “2*n + 1”

For example, if we assume that:

- each disk block that contains part of the Lectures relation contains several tuples, and
- each request to find a matching row in the Courses relation requires one block to be brought in from disk, and
- for each tuple in the Lectures relation retrieving the matching tuple from the Courses relation requires one block to be brought in from disk

and/or

- the entire index is stored across many disk blocks

Then the cost will be greater than 2*n + 1

(New: 2020-05-31)

Cost can be less than “2*n + 1”

For example, if we assume that:

- several tuples in the Lectures relation match with the same tuple in the Courses relation, and that the block containing this tuple is still in main memory when subsequent access requests are made,

and/or

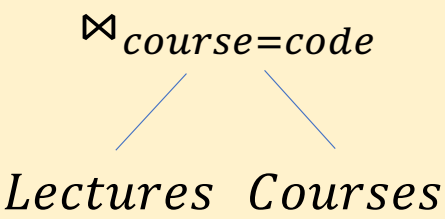
- each disk block that contains part of the Courses relation contains several tuples

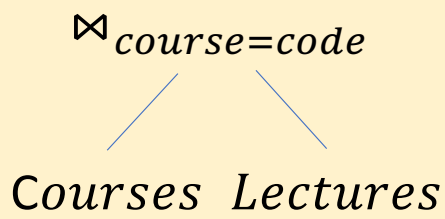
Then it will not be necessary to read a new disk block from the Courses relation into main memory for every tuple in the Lectures relation, and the cost can be less than 2*n + 1

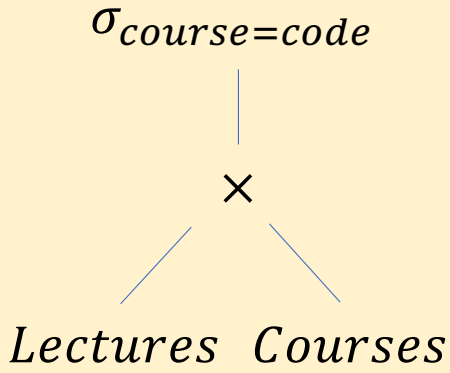
(New: 2020-05-31)

Query plan

- In the previous slides we have assumed that the SQL query will be translated to the query plan on the left.
- The query processor might instead choose another equivalent query plan that has a different cost.







(New: 2020-05-31)

Cold database vs. warm database

- Sometimes running the same query a second time will be much faster than running the query the first time.
- If we assume that there are initially no disk blocks in main memory (i.e. the database is “cold”), then all disk blocks that are required to answer the query will be brought into main memory (but will not necessarily be in main memory at the same time) when the query is executed.
- If we now run the same query again, we might find that some of the required disk blocks are already in main memory from the previous run (i.e. the database is “warm”), and so don’t need to be brought in from disk again.
- When benchmarking database performance it is important to know whether the test queries are run against a cold or warm database.

(New: 2020-05-31)

“2*n + 1” is a rough indication of query cost

The actual cost of executing the query when there is an index on code in Courses is almost certainly not exactly equal to 2*n + 1.

Nevertheless, it is useful to know that cost of the query plan is generally linear with the size of the Lectures relation, plus a bit for reading the index.

Assuming m > 2, we would expect that n*m (which is also a rough approximation) will be larger than 2*n + 1.

It is common that rewriting a query plan can change the cost from being, say, n^2 to $n.\log(n)$ or linear. It is these big differences that are important for query optimisation.

(New: 2020-05-31)

CREATE INDEX

- Most relational 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).

Indexes in a RDBMS

- 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.

Why don’t we have indexes on all 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.

Lectures(course, period, weekday, hour, room)

How costly are these queries?

Assume we have an index on Lectures for (course, period, weekday) which is the key.

Lectures: n disk blocks

```
SELECT *                    SELECT *
FROM   Lectures             FROM   Lectures
WHERE  course = 'TDA356'    WHERE  weekday = 'Monday'
       AND period = 2;             AND room = 'VR';
```

A multi-attribute index is typically organized hierarchically. First the rows are indexed according to the first attribute, then according to the second within each group, and so on.
Thus the ***left*** query costs at most ***k + 1*** where k is the number of rows matching the values. The ***right*** query can't use the index, and thus costs ***n***, where n is the size of the relation in disk blocks.

Lectures(course, period, weekday, hour, room)

Example: Suppose that the Lectures relation is stored in 20 disk blocks, and that we typically perform three operations on this table:

- insert new lectures (Ins)
- list all lectures of a particular course (Q1)
- list all lectures in a given room (Q2)

Let's assume that in an average week there are:

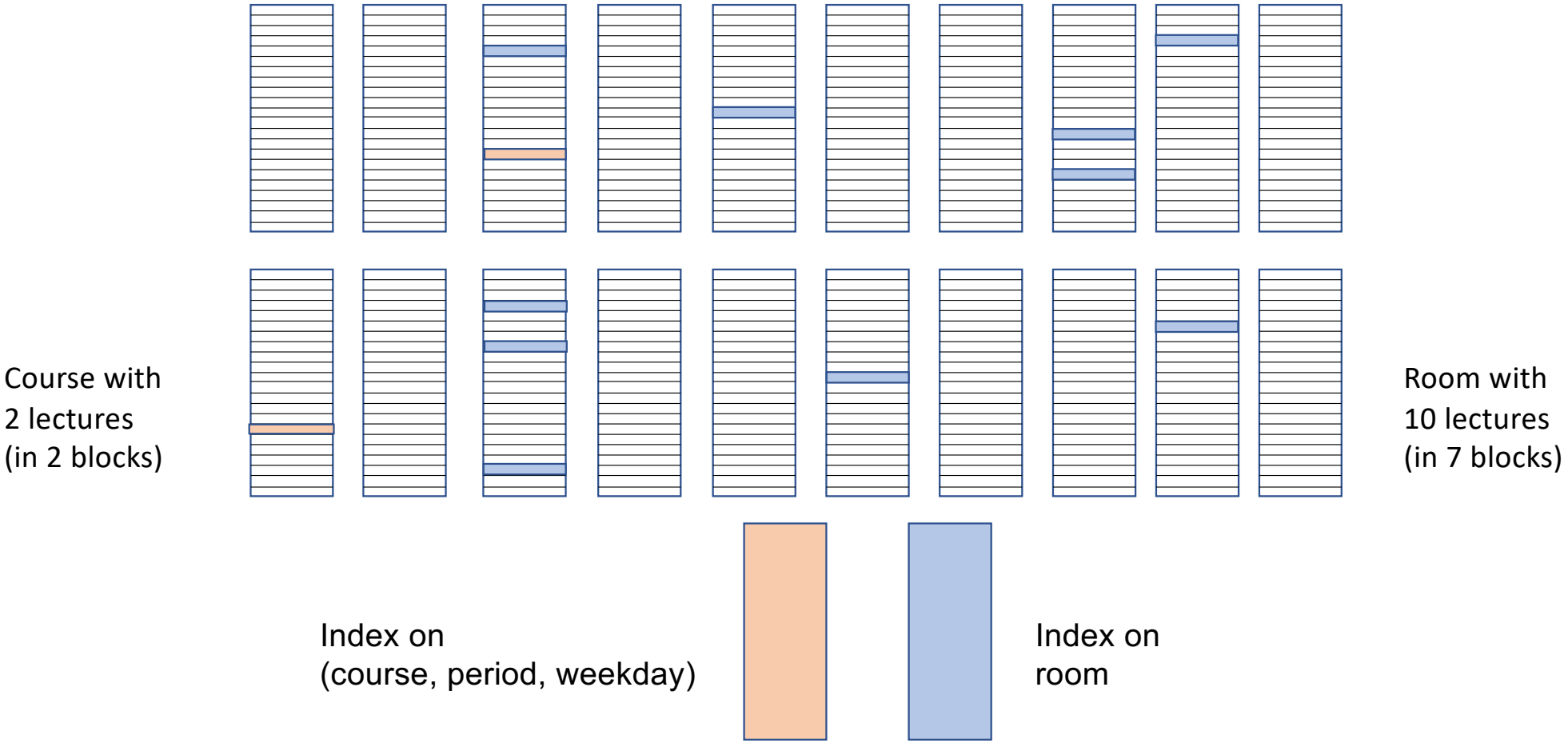
- 2 lectures for each course, and
- 10 lectures in each room.

Let's also assume that

- each course has lectures stored in 2 blocks, and
- each room has lectures stored in 7 (some lectures are stored in the same block).

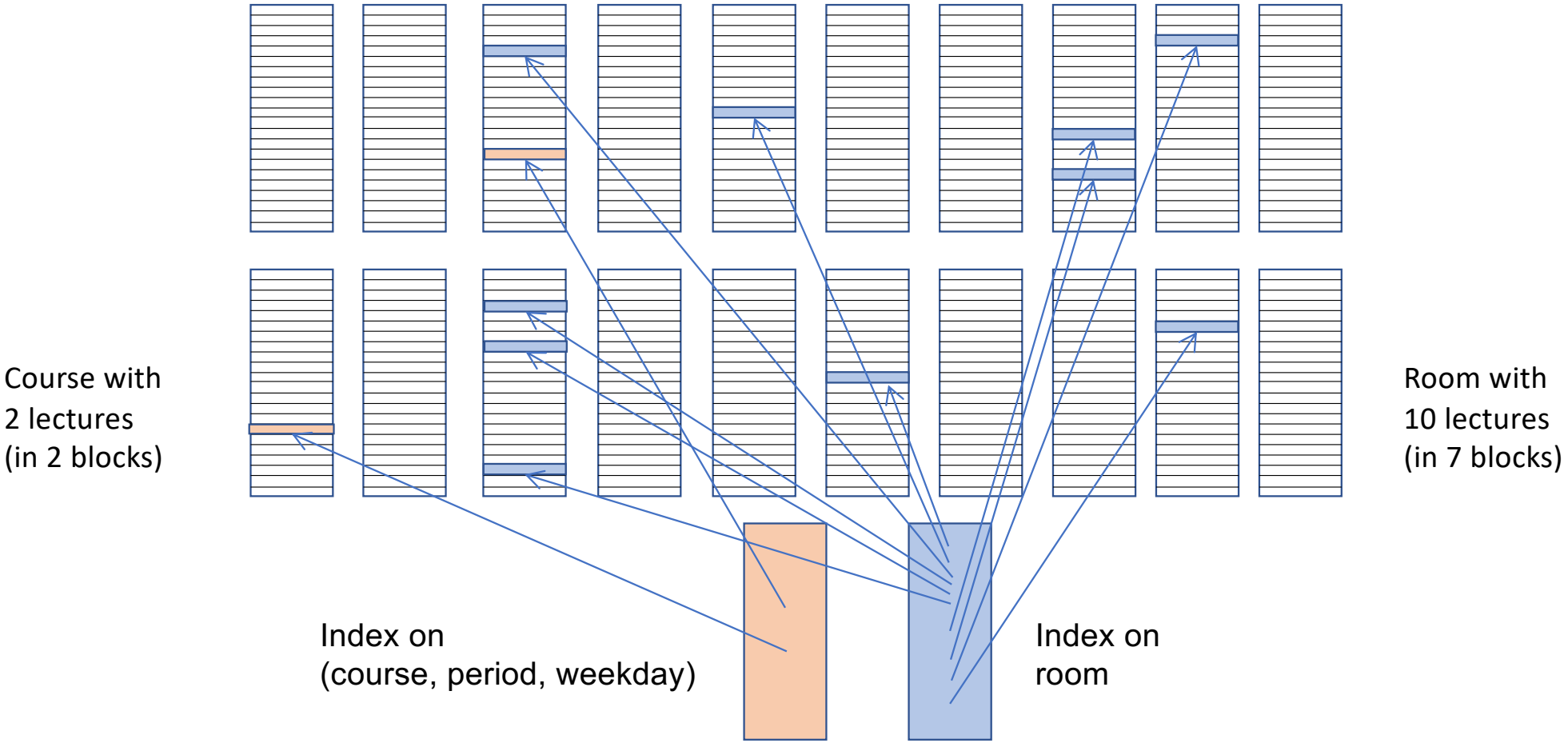
Lectures(course, period, weekday, hour, room)

Lectures example: disk blocks



Lectures(course, period, weekday, hour, room)

Lectures example: disk blocks



Lectures(course, period, weekday, hour, room)

Costs

Insert new lectures (Ins)
List all lectures of a particular course (Q1)
List all lectures in a given room (Q2)

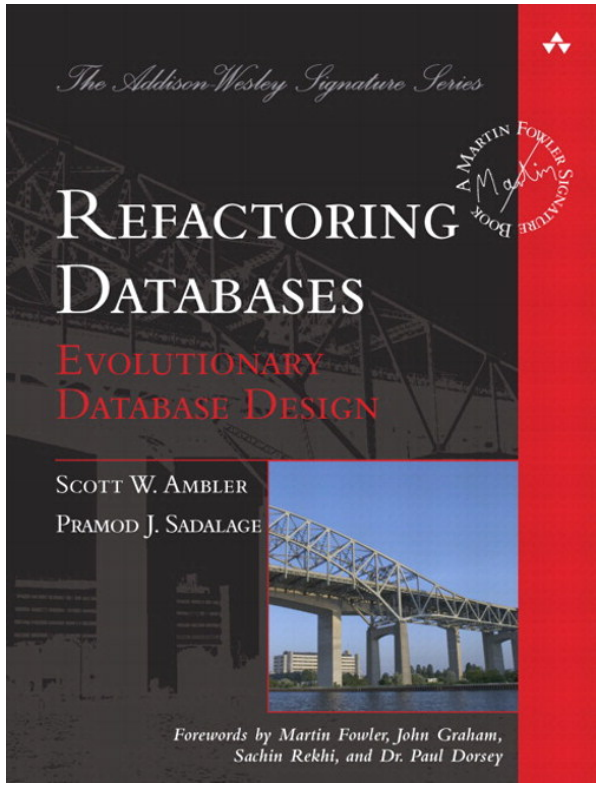
	Case A	Case B	Case C	Case D
	No index	Index on (course, period, weekday)	Index on room	Both indexes
Ins	2	4	4	6
Q1	20	3	20	3
Q2	20	20	8	8

The amortized cost depends on the proportion of operations of each kind.

Ins	Q1	Q2	Case A	Case B	Case C	Case D
0.2	0.4	0.4	16.4	10	12	5.6
0.8	0.1	0.1	5.6	5.5	6	5.9
0.1	0.6	0.3	18.2	8.2	14.8	4.8

Refactoring databases

- Small changes to table structures, data, stored procedures, and triggers can significantly improve system designs, maintainability, extensibility, and performance
- How to evolve relational database schemas without changing semantics
- Tradeoffs



Refactoring example 1

Replace a natural key by a surrogate key

Motivation:

- **Reduce coupling between table schema and business domain**. If part of a key is likely to change (in size or type), it's a bad idea to have it as a primary key.
- **Increase consistency**. Potentially improve performance and reduce code complexity.
- **Improve database performance**. Some RDBMSs struggle with large composite natural keys; updating the index is difficult.

Refactoring example 2

Replace a surrogate key by a natural key

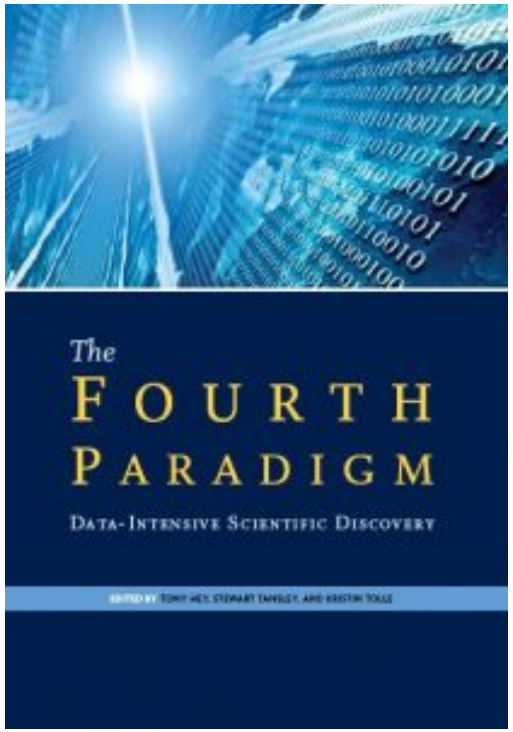
Motivation:

- **Reduce overhead**. No need to maintain additional surrogate key column(s).
- **Consolidate key strategy**. Improves code consistency
- **Remove non-required keys**. You might discover that a surrogate key is not really needed, and removing unused indexes will improve performance.

The Fourth Paradigm

"The Fourth Paradigm: Data-Intensive Scientific Discovery", Edited by Tony Hey, Stewart Tansley, and Kristin Tolle

1. Empirical
   describing natural phenomena
2. Theoretical
   using models, generalisations
3. Computational
   simulating complex phenomena
4. **Data exploration**



https://www.microsoft.com/en-us/research/publication/fourth-paradigm-data-intensive-scientific-discovery/

“20 queries”

- ”Jim Gray came up with the heuristic rule of “20 queries.” On each project he was involved with, he asked for the 20 most important questions the researchers wanted the data system to answer. He said that five questions are not enough to see a broader pattern, and a hundred questions would result in a shortage of focus.”

"Gray’s laws: database-centric computing in science" by Alexander S. Szalay and José A. Blakeley, pp 5-11. In "The Fourth Paradigm: Data-Intensive Scientific Discovery", Edited by Tony Hey, Stewart Tansley, and Kristin Tolle

Semantic data modelling

Semantic data models represent entities and the relationships between them directly, and match closely our conceptual view of data.

Uses three “abstraction mechanisms”:

- **Classification**

  entities which share common characteristics are grouped together into instances of a class

- **Aggregation**

  regard a collection of values as properties of a single compound object or aggregate

- **Generalisation**

  if two or more classes have characteristics in common, then these commonalities can be abstracted into a general class

Some videos on RDF, RDFS and SPARQL

”GraphDB Fundamentals - Module 1: Overview of RDF & RDFS”
- https://www.youtube.com/watch?v=iuQrBf2Oq-E

”GraphDB Fundamentals - Module 2: Overview of SPARQL”
- https://www.youtube.com/watch?v=L_eB7Z84M4c

”SPARQL in 11 minutes”
- https://www.youtube.com/watch?v=FvGndkpa4K0

”03 - 05 How to Store RDF(S) Data? - Triple Stores”
- https://www.youtube.com/watch?v=Dxwo9DYWV_c

”Using DBpedia”
- https://www.youtube.com/watch?v=BmHKb0kLGtA

Wikidata Sparql Query Tutorial
- https://www.youtube.com/watch?v=1jHoUkj_mKw

On Wednesday we’ll look at the Semantic Web. I recommend watching (at least the first 4 of) these short videos before the classes on Wednesday.

You can try using DBpedia, Wikidata and GraphDB at the lab on Wednesday (or before).
Some instructions for getting started will be added to Canvas.