INFO-H417 - Database System Architecture

Synthesis

Answers/theory for the written exam topic questions (based on oral course, slides, PostgreSQL documentation, ChatGPT and internet)

ACHTEN Alexandre and HUMBLET Raphaël
15/01/2024

Table of contents

1 Translating SQL into Relational Algebra

1.1 Express SQL queries with selection, projection, join, aggregation, CTE and subquery

First, let's recap some vocabulary. **Relations** are **tables** whose **columns** have names, called **attributes**. The set of all attributes of a relation is called the **schema of the relation**. The **rows** in a relation are called **tuples**.

1.1.1 Selection and projection

SQL allows the user to produce some queries. There are generally of the form:

```
SELECT attributes
FROM table
WHERE conditions
```

Here we can see **selection** with the **WHERE** condition and **projection** with the **SELECT** clause.

1.1.2 Join

We can also use **join** to match two relations.

```
SELECT attributes
FROM table1, table2
WHERE table1.common_attributes = table2.common_attributes
```

1.1.3 Aggregation

We can use **aggregation** thanks to the **GROUP BY** clause and apply conditions on it thanks to the **HAVING** clause.

```
SELECT department, AVG(age) as avg_age
FROM students
GROUP BY department
HAVING AVG(age) > 25;
```

The aggregate functions include **COUNT**, **SUM**, **AVG**, **MIN** and **MAX**.

1.1.4 CTE – Common Table Expression

We talk about **CTE (Common Table Expression)** for temporary result set within the scope of an SQL query. It allows to define a query expression and reference it within the context of a larger SQL statement. It is defined using the **WITH** clause in SQL queries.

```
WITH CTE_name (column1, column2, ...) AS (
  -- Query expression for CTE
  SELECT ...
)
SELECT ...
FROM CTE_name;
```

CTEs provide a way to simplify complex queries, break them into modular pieces, and make the SQL code more readable and maintainable.

1.1.5 Subqueries

We can also talk about **subqueries**, these are used to perform a

```
SELECT column1, column2, ...
FROM table1
WHERE column_name operator (
    SELECT column_name
    FROM table2
    WHERE ...);
```

We can always **normalize** subqueries to **use only EXISTS and NOT EXISTS**.

This allows to flatten the subquery. Note that subqueries are not always the best option as it forces the program to execute the subquery for each element. Therefore sometimes, the optimiser will try to **flatten the subquery** (see Section 1.6).

1.2 Describe the concepts of declarative and procedural query languages

1.2.1 Declarative Query Language:

A **declarative query language** is a type of programming language where you **specify the desired outcome or result**, and the **system determines the best way to achieve that result**.

In a declarative language, **you express what you want to achieve**, but you **don't explicitly specify how to achieve it**. Instead, the system's query processor or optimizer interprets your query and determines the most efficient way to execute it.

SQL (Structured Query Language) is one of the most well-known examples of a declarative query language, commonly used for interacting with relational databases. In SQL, you write queries that describe the data you want to retrieve or manipulate, and the database engine takes care of the details of how to execute those queries.

1.2.2 Procedural Query Language:

A **procedural query language**, on the other hand, **requires you to specify the step-by-step procedure or sequence of operations to achieve the desired outcome**. In a procedural language, you explicitly describe the algorithm or logic that the system must follow to execute the query.

Programming languages like **Python**, **Java**, or **C** are procedural languages. In the context of databases, procedural languages are often used in stored procedures, functions, or triggers. Stored procedures, for example, contain a series of statements that are executed in a specific order to perform a certain task.

1.3 Describe the different relational algebra (and extended RA) operators

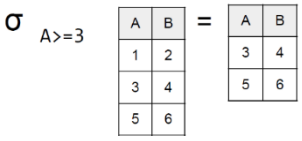
Operator		Operator	
$\cup$	Union	$\bowtie$	Natural join
$\cap$	Intersection	$\bowtie_{B=C}$	Theta join
-	Difference	$⟕_{B=C}$	Left outer join
$\sigma_{A>=3}$	Selection	$⟖_{B=C}$	Right outer join
$\pi_{A,C}$	Projection	$⟗_{B=C}$	Full outer join
$\times$	Cartesian product	$\gamma_{A,min(B)->D}$	Aggregation
ρ_T	Rename	$\leftarrow$	Assignment

15

Figure 1: RA and extended RA operators

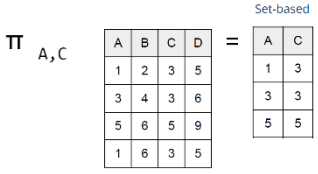
Selection in RA consists in applying a condition on a relation. It will **select the rows/tuples of the relation that satisfy the condition**.

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



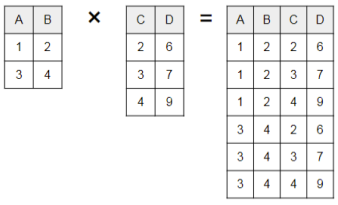
Projection in RA consists in **selecting certain columns/attributes of a relation**. Note that the result of a projection is always **set-based**.

$$\pi_{attribute_1,\ldots,attribute_n}(relation)$$



A **cartesian product** in RA allows to **match all tuples from a relation to all tuples of another relation**. Note that the two relations need to have disjoint set of attributes, otherwise we need to rename them first (There cannot be the same attribute name in both relation).

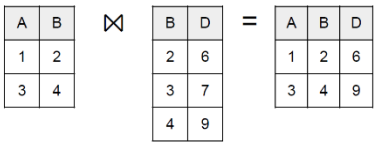
$$relation_1 \times relation_2$$



The **natural join** operation in RA will act differently depending on the relations concerned by it.

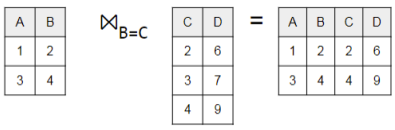
$$relation_1 \bowtie relation_2$$

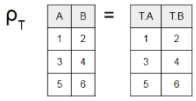
- **If the two relations have disjointed schema**, it will act as a **cartesian product**.
- Otherwise, it will match all tuples of the first relation with all the tuples of the other relation **based on a common attribute** (here on the example, B is the common attribute).



The **theta join** is a natural join combined with a condition θ.

$$relation_1 \bowtie_\theta relation_2$$

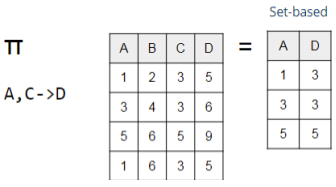




The **rename** operator in RA allows to rename a relation. $\rho_{newName}(relation)$

The **extended projection** allows renaming to while keeping all the properties of a normal projection.

$$\pi_{attribute_1,attribute_2 \to newAttribute_2}(relation)$$



The **grouping** operator allows to execute aggregation functions such as **COUNT**, **SUM**, **AVG**, **MIN** and **MAX**.

$$\gamma_{attribute_1,FUN(attribute_2)\rightarrow newAttribute}(relation)$$

$\gamma_{A,min(B)->D}$

A	B	C
1	2	a
1	3	b
2	3	c
2	4	a
2	5	a

=

A	D
1	2
2	3

There are also the **union** ∪, **intersection** ∩ and **difference** − operators. Note that those needs that both relations have the **same schema** (set of attributes).

A	B
1	2
3	4
5	6

∪

A	B
3	4
1	5

=

Set-based

A	B
1	2
3	4
5	6
1	5

Bag-based

A	B
1	2
3	4
5	6
1	5
3	4

It can give two different outputs depending on if it is **set-based** or **bag-based**.

1.4 Translate a given SQL statement into an equivalent RA expression

Here are some examples:

```
SQL:
SELECT movieTitle
FROM   StarsIn S, MovieStar M
WHERE  S.starName = M.name AND M.birthdate = 1960
```

RA ?

$$\pi_{movieTitle}\ \sigma_{S.starName=M.name\ and\ M.birthdate=1960}(\rho_S(StarsIn) \times \rho_M(MovieStar))$$

```
SQL:
SELECT movieTitle, count(S.startName) AS numStars
FROM   StarsIn S, MovieStar M
WHERE  S.starName = M.name
GROUP BY movieTitle
```

RA ?

$$\gamma_{M.movieTitle,count(S.starName)->numStars}($$
$$\rho_S(StarsIn) \bowtie_{S.starName=M.name} \rho_M(MovieStar))$$

```
SELECT movieTitle, count(S.startName) AS numStars
FROM   StarsIn S, MovieStar M
WHERE  S.starName = M.name
GROUP BY movieTitle
HAVING count(S.startName) > 5
```

RA ?

$$\sigma_{numStarts>5}(\gamma_{M.movieTitle,count(S.starName)->numStars}($$
$$\rho_S(StarsIn) \bowtie_{S.starName=M.name} \rho_M(MovieStar)))$$

1.5 Differentiate the set and bag semantics in RA, and describe how the query results may differ applying one semantic or the other

A relation is **set-based** if it **does not contain duplicate tuples**. It is called **bag-based** otherwise.

Therefore, a set-based query will produce a concise result without duplicate tuples while a bag-based query will provide with a more detailed view of the data and will be useful when the frequency or occurrence of certain tuples is significant for analysis.

Set-based query:

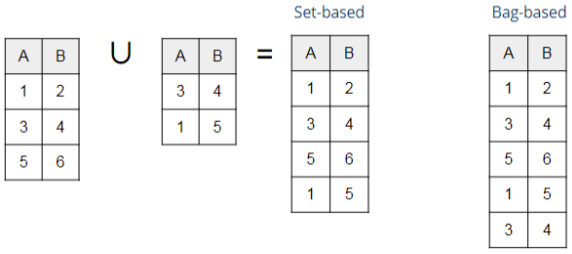
```
SELECT A, B
FROM Relation1
UNION
SELECT A, B
FROM Relation2;
```

In set semantics, if both Relation1 and Relation2 contain a tuple with the same values for attributes A and B, the result **will include only one copy of that tuple**.

Bag-based query:

```
SELECT A, B
FROM Relation1
UNION ALL
SELECT A, B
FROM Relation2;
```

In bag semantics, the **result will include all tuples from both relations, even if there are duplicates**.



1.6 Illustrate the transformation of a sub-query into a join then into an RA expression

The problem with **subqueries** is that it blocks the optimiser. So, to avoid this, there are procedures to **flatten the subqueries into normal join** so that the *optimiser can have more options to optimally execute the query*.

Before translating a query, we first **normalise** it such that all the subqueries that occur in a where condition are of the form **EXISTS** or **NOT EXISTS**.

Correlated Subquery can refer to attributes of relations that are introduced in an outer query.

```
SELECT movieTitle
FROM StarsIn S
WHERE EXISTS (SELECT name
              FROM MovieStar
              WHERE birthdate=1960 AND name=S.starName)
```

Here for example, we use *name=S.starName* in the subquery even though it is defined in the outer query (main query).

→ The **set of all attributes of all context relations of a subquery** are called the **parameters of the subquery**.

To **translate it into RA expression**, we need to do a **flattening** of the subquery:

1. Translate the subquery
2. Add the context relation and parameters
3. Translate the FROM clause of the outer query
4. Synchronise both expressions by means of a join
5. Simplify
6. Finally, complete the expression.

But note that this is **only applicable to set-based relations** (relation that does not contain redundant tuples).

Example of **flattening into a normal join for set-based relations**:

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND EXISTS (SELECT name
            FROM MovieStar
            WHERE birthdate=1960 AND name= S.starName)
```

1. Subquery translation

$$\pi_{Name}(\sigma_{birthdate=1960 \wedge name=S.starName}(MovieStar))$$

(S.starName: parameter)

2. Add context relation and parameter

$$\pi_{Name, S.movieTitle, S.movieYear, S.starName}(\sigma_{birthdate=1960 \wedge name=S.starName}(MovieStar \times \rho_S(StarsIn)))$$

3. Translate the FROM clause of the outer query

$$\rho_S(StarsIn) \times \rho_M(Movie)$$

4. Synchronize both expression with a join

$$\rho_S(StarsIn) \times \rho_M(Movie) \bowtie (\pi_{Name, S.movieTitle, S.movieYear, S.starName}(\sigma_{birthdate=1960 \wedge name=S.starName}(MovieStar \times \rho_S(StarsIn))))$$

5. Simplify (...)

$$\rho_M(Movie) \bowtie (\pi_{S.movieTitle, S.movieYear, S.starName}(\sigma_{birthdate=1960 \wedge name=S.starName}(MovieStar \times \rho_S(StarsIn))))$$

6. Complete the expression

$$\pi_{S.movieTitle, M.studioName}(\sigma_{S.movieYear>=2000 \wedge S.movieTitle=M.title}(\rho_M(Movie) \bowtie (\pi_{S.movieTitle, S.movieYear, S.starName}(\sigma_{birthdate=1960 \wedge name=S.starName}(MovieStar \times \rho_S(StarsIn))))))$$

To do a **flattening to a normal join** of a subquery in **bag-based relations**, there are requirements on the subquery:

- The subquery is *not under an OR*.
- The subquery *type is EXISTS, IN, or ANY*, or it is an *expression subquery on the right side of a comparison operator*.
- The subquery is *not in the SELECT list of the outer query block*.
- There are *no aggregates in the SELECT list of the subquery*.

- The subquery *does not have a GROUP BY clause*.
- The subquery *does not have an ORDER BY*, *result offset*, or *fetch first clause*.
- There is a **uniqueness condition** that *ensures that the subquery does not introduce any duplicates* if it is flattened into the outer query block.
- Each table in the subquery's FROM list (after any view, derived table, or subquery flattening) must be a base table.
- If there is a WHERE clause in the subquery, there is at least one table in the subquery whose columns are in equality predicates with expressions that do not include any column references from the subquery block. These columns must be a superset of the key columns for any unique index on the table. For all other tables in the subquery, the columns in equality predicates with expressions that do not include columns from the same table are a superset of the unique columns for any unique index on the table.

For more information:
https://docs.oracle.com/javadb/10.8.3.0/tuning/ctuntransform36368.html.

2 System R & Query Optimizations

https://www.seas.upenn.edu/~zives/cis650/papers/System-R.PDF

Firstly, **System R** is an experimental database system that was developed to explore and pioneer concepts of relational databases. Mainly, it provides high-level, data-independent facilities for data retrieval, manipulation, and definition through the RDI (see Section 2.1).

2.1 Describe the architecture components of system R

The **System R** is composed of 3 levels: the user level, the data level, and the storage level.

The **Relational Storage Interface (RSI)** is an **internal** **interface** which **handles access to single tuples of base relations**. This interface and **its supporting system**, the **Relational Storage System (RSS)**, is a complete storage subsystem that **manages devices, space allocation, storage buffers, transaction consistency and locking, deadlock detection, backout, transaction recovery, and system recovery**. Furthermore, it maintains indexes on selected fields of base relations, and pointer chains across relations.

The Relational Data Interface (RDI) is the **external** **interface** for **high-level, data-independent facilities for data retrieval, manipulation, definition, and control**. It can be called directly from a programming language or used to support various emulators and other interfaces. The **Relational Data System (RDS),** which **supports the RDI**, provides **authorization, integrity enforcement, and support for alternative views of data**. It is where the **relational algebra takes place**. The RDS **contains an optimizer** which **plans the execution of each RDI command, choosing a low-cost access path to data from among those provided by the Relational Storage System (RSS).**

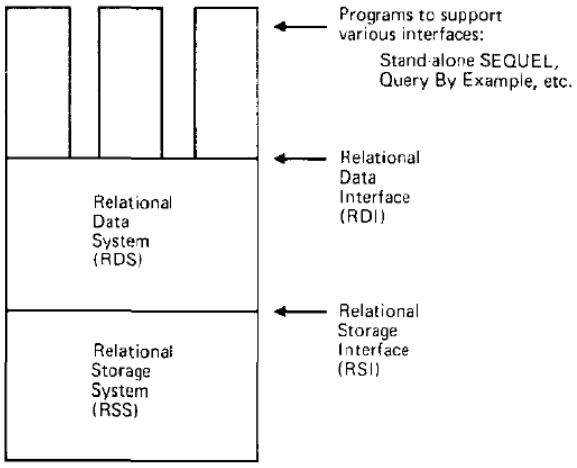


FIG. 1. Architecture of System R

The high-level **SEQUEL language** is *embedded within the RDI*. It is used as the basis for all data definition and manipulation.

The **Relational Data Interface** (RDI) is the principal external interface of System R. The data definition facilities of the RDI allow a **variety of alternative relational views** to be defined **on common underlying data**.

The objective of the **optimiser** (within the RDS) is *to find a low cost means of executing a SEQUEL statement*, given the data structures and access paths available. It is part of the RDS.

- The optimiser attempts to **minimize the expected number of pages to be fetched** from secondary storage into the RSS buffers during execution of the statement. Only page fetches made under the explicit control of the RSS are considered.

- **The cost of CPU instructions is also considered by means of an adjustable coefficient, H, which is multiplied by the number of tuple comparison operations to convert to equivalent page accesses**.
- H can be adjusted according to whether the system is compute-bound, or disk access bound. Since our cost measure for the optimiser is based on disk page accesses, the physical clustering of tuples in the database is of significant importance.

2.2 Concepts of System R and PostgreSQL implementation

In this section, the concepts are defined for System R and then, their implementation in PostgreSQL is discussed.

2.2.1 Catalog

Catalog: The catalog of a database in System R is a **repository or a set of tables that stores metadata** about the database. It *contains information about tables, columns, indexes, constraints, users, and other database objects*. The catalog provides a **systematic way to organize and manage information** about the database schema and its components. It enables the database management system (DBMS) to interpret and execute queries, maintain data integrity, and manage the overall database structure. It also serves to get information about the **system statistics**.

→ **PostgreSQL** has a system catalog that *stores metadata about the database*. This includes **information about tables**, **columns**, **indexes**, **constraints**, **materialized views**, **optimization rules**, **SQL user functions**, **user defined types** and the **database schema**.

Note that the catalog contains the information about those but NOT the data itself. It is mostly the i/o functions, types, etc that are stored in the catalog. For more information, check PostgreSQL documentation.

One particularity of it is that users can query system's catalog to retrieve information about the database schema. It is a table that is called **pg_statistics**.

2.2.2 Tuple identifier

Tuple identifier (TID): A tuple identifier is a **unique identifier assigned to each tuple** (row) in a relation (table) within the database. It serves to uniquely identify and access individual rows. Tuple identifiers serve for efficient data retrieval and manipulation. They are used in indexing, storage, and retrieval operations, ensuring that each tuple can be uniquely referenced.

→ **PostgreSQL** uses a system-generated unique identifier known as the "oid" (object identifier) for each row in a table. Additionally, *primary keys or unique constraints are often used to identify tuples uniquely*.

2.2.3 Image

Image: In the context of System R, an image refers to a snapshot or a **representation of the database at a specific point in time**. It captures the state of the database, including the values of tuples and the database schema, **at a particular moment**. There can be several images for the same relation as they are all linked to a certain snapshot/moment in time. But there cannot be multiple images for a single attribute as those are too small.

Images are used for various purposes, such as supporting transactions, enabling database recovery, and providing a consistent view of data during query execution.

→ **PostgreSQL** maintains **a transaction log** (**WAL** - Write-Ahead Logging) that **allows for database recovery**. It captures changes to the database and provides a consistent state (image) for recovery purposes.

2.2.4 Clustering image

Clustering Image: This has the property that tuples near each other in the image ordering **are stored physically near each other in the database**. This allows to **gain some I/O time** as all related tuples are stored in groups/cluster. In general, a cluster contains at least 20 adjacent tuples, thus 1 IO allows to get 20 tuples!

➔ If the image is not the clustering image, the locations of the tuples will be independent of each other and in general, a page will have to be fetch on disk **for each tuple**.

→ **PostgreSQL** implements **clustering** by using index as in the following:

CLUSTER table_name USING index_name

Command that physically reorders the table based on the specified index. Note that the index is generally a **BRIN** (Block Range INdex) as it allows for the organisation of data in blocks/range. But other indexes can work too.

PostgreSQL supports clustering tables **based on an index**. This physically reorganizes the table's rows to match the order of the specified index, improving query performance for certain types of queries.

2.2.5 View

View: A **view** in System R is a **virtual table that is derived from one or more base tables**. It represents a predefined query stored in the catalog, and its result set is dynamically computed when the view is queried. Views provide a way to simplify complex queries, encapsulate business logic, and restrict access to certain columns or rows.

→ **PostgreSQL** supports views, which are **virtual tables derived from one or more base tables**. Views provide a way to encapsulate complex queries and offer a simplified, abstracted layer over the underlying tables.

2.2.6 Cost-based query optimization

Cost-based query optimization: Cost-based query optimization is a **method used by the query optimizer (within RDS) to select the most efficient execution plan** for a given query **based on estimated costs**.

The optimizer evaluates various possible execution plans and chooses the one with the lowest estimated cost. The goal is to minimize the overall resource usage (e.g., CPU, I/O) and improve query performance. Cost-based optimization involves ***estimating*** *the cost of different access paths, join algorithms, and other factors influencing query execution*.

Steps in cost-based query optimization:

1. Generate logically equivalent expressions **using equivalence rules**
2. Annotate resultant expressions to get alternative query plans
3. **Choose the cheapest plan based on estimated cost**

Estimation of plan cost based on:

- Statistical information about relations. Examples: number of tuples, number of distinct values for an attribute
- Statistics estimation for intermediate results to compute cost of complex expressions
- Cost formulae for algorithms, computed using statistics

→ **PostgreSQL** employs a cost-based query optimizer that evaluates different execution plans based on estimated costs. The optimizer **considers factors such as available indexes, join algorithms, and statistics to choose an efficient access plan**.

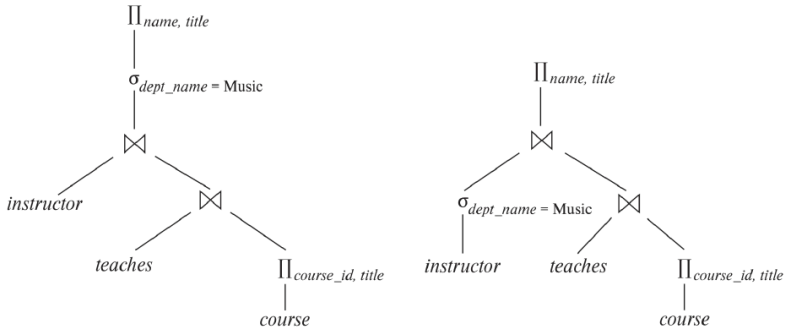
2.2.7 Access path

Access path: An access path refers to the mechanism or **route used by the database system to access data during query execution**. In **RSS**, but defined in **RDS**. It includes strategies for navigating and retrieving data, such as indexes, full table scans, or clustering. The choice of an appropriate access path significantly impacts query performance. The query optimizer selects an access path based on factors like available indexes, data distribution, and the estimated cost of different access methods.

→ **PostgreSQL** provides various access paths for query execution. This includes **indexes** (B-tree, GIN, GiST, etc.), sequential scans, bitmap scans, and other methods. The query planner analyses the query and selects the most efficient access path based on cost estimates.

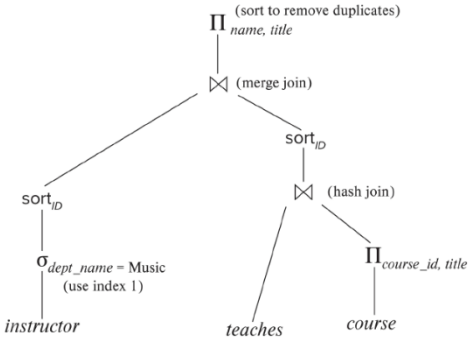
2.3 Query optimization

There are alternative ways of evaluating a given query, for example using equivalent expressions or different algorithms for each operation.



(a) Initial expression tree (b) Transformed expression tree

Query plan: An evaluation plan **defines** exactly **what algorithm is used for each operation**, and **how** the **execution** of the operations **is coordinated**. It also helps the optimiser estimating the cost of the query.



Query optimizer: Query optimizers **use equivalence rules** to systematically **generate expressions equivalent to** the **given expression**.
It generate all equivalent expressions as follows:

- Repeat
  - Apply all applicable equivalence rules on every subexpression of every equivalent expression found so far
  - Add newly generated expressions to the set of equivalent expressions.
- Until no new equivalent expressions are generated above.

Must consider the interaction of evaluation techniques when choosing evaluation plans. Choosing the cheapest algorithm for each operation independently may not yield best overall algorithm. E.g. merge-join may be costlier than hash-join but may provide a sorted output which reduces the cost for an outer level.

Practical query optimizers incorporate elements of the following **two broad approaches**:

1. Search all the plans and choose the best plan in a cost-based fashion.
2. Uses heuristics to choose a plan.

2.4 Application of equivalence rules

Note that the rules do not need to be memorised as they will be given on the exam if necessary.

Two relational algebra expressions are said to be equivalent if the two expressions generate the same set of tuples on every legal database instance. In SQL, inputs and outputs are multisets (bag) of tuples. An **equivalence rule** says that expressions of two forms are equivalent, so one can replace the other.

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

$$\sigma_{\theta 1 \wedge \theta 2}(E) \equiv \sigma_{\theta 1}(\sigma_{\theta 2}(E))$$

2. Selection operations are commutative.

$$\sigma_{\theta 1}(\sigma_{\theta 2}(E)) \equiv \sigma_{\theta 2}(\sigma_{\theta 1}(E))$$

3. Only the last in a sequence of projection operations is needed, the others can be omitted (where L1 ⊆ L2 ... ⊆ Ln)

$$\Pi_{L1}(\Pi_{L2}(\ldots(\Pi_{Ln}(E))\ldots)) \equiv \Pi_{L1}(E)$$

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

$$\sigma_{\theta}(E_1 \times E_2) \equiv E_1 \bowtie_{\theta} E_2$$

$$\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) \equiv E_1 \bowtie_{\theta 1 \wedge \theta 2} E$$

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

$$E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$$

6. (a) Natural join operations are associative:

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

(b) Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 \equiv E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$

where θ_2 involves attributes from only E_2 and E_3

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

(a) When all the attributes in θ_0 involve only the attributes of one of the expressions (E1) being joined.

$$\sigma_{\theta 0}(E_1 \bowtie_{\theta} E_2) \equiv (\sigma_{\theta 0}(E_1)) \bowtie_{\theta} E_2$$

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

$$\sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_{\theta} E_2) \equiv (\sigma_{\theta 1}(E_1)) \bowtie_{\theta} (\sigma_{\theta 2}(E2))$$

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

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

$$\Pi_{L1 \cup L2}(E_1 \bowtie_{\theta} E_2) \equiv \Pi_{L1}(E_1) \bowtie_{\theta} \Pi_{L2}(E_2)$$

Similar equivalence hold for outerjoin operations: ⟕, ⟖, and ⟗

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 \equiv E_2 \cup E_1 \qquad E_1 \cap E_2 \equiv E_2 \cap E_1$$

(set difference is not commutative).

10. Set union and intersection are associative.

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

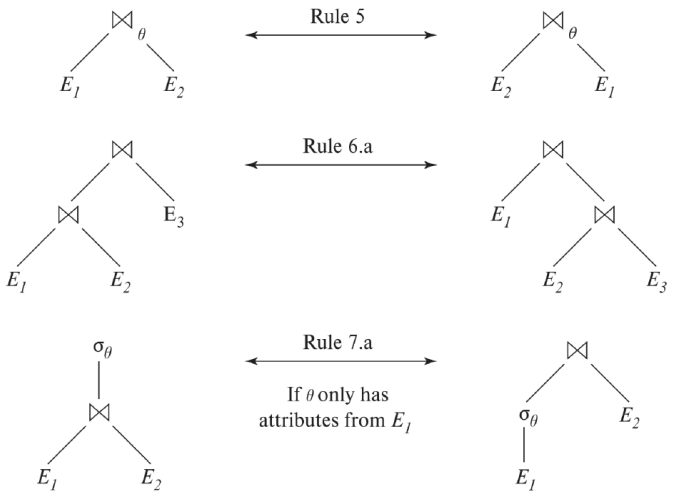
11. The selection operation distributes over $\cup$, $\cap$ and $-$.

$$\sigma_\theta (E_1 \cup E_2) \equiv \sigma_\theta (E_1) \cup \sigma_\theta(E_2)$$
$$\sigma_\theta (E_1 \cap E_2) \equiv \sigma_\theta (E_1) \cap \sigma_\theta(E_2)$$
$$\sigma_\theta (E_1 - E_2) \equiv \sigma_\theta (E_1) - \sigma_\theta(E_2)$$
$$\sigma_\theta (E_1 \cap E_2) \equiv \sigma_\theta(E_1) \cap E_2$$
$$\sigma_\theta (E_1 - E_2) \equiv \sigma_\theta(E_1) - E_2 \text{ (does not hold for } \cup)$$

12. The projection operation distributes over union

$$\Pi_L(E_1 \cup E_2) \equiv (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

Here is an example:



2.5 Illustrate the computational challenge of cost-based query optimization

The biggest challenge in cost-based query optimization is that we need to **compute all query plans** and compare them to take the best solution. This can lead to outrageous computations as seen in the join ordering problem (see next Section 2.6).

To get some numbers, let's take this join ordering case:

$$r_1 \bowtie r_2 \ldots \bowtie r_n$$

There are $\frac{(2*(n-1))!}{(n-1)!}$ possible cases: for $n = 10$, this means 176 billions! Therefore, there are solutions/hypotheses that will be done in order to reduce this number.

Another problem that could occur is that the order of execution of the queries will change a lot the cost of a query.

Some solutions are **dynamic programming** with **memoisation** (storing in a cache the already computed solutions so that they can be reused for other query plans) or **Heuristics** to diminish the number of query plans for which we will compute the cost.

Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10).

Note that the optimiser has a **budget time**, if it goes past this time limit, it will give the best answer that has been found until then.

2.6 The join order optimization problem

The **join order optimization problem** arises when executing queries involving multiple tables, and it involves **determining the most efficient order** in which to perform joins.

$$r_1 \bowtie r_2 \ldots \bowtie r_n$$

The order of joins can **significantly impact the overall query performance**, as different join orders may result in different intermediate results and execution times.

There are way too much query plans possible and it would take way too long to compute the cost of all of them.

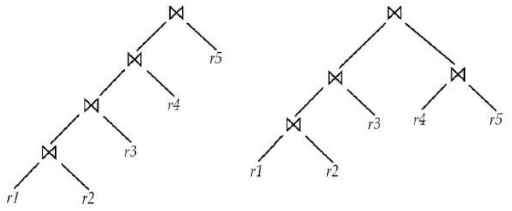
Let's use **dynamic programming** to help reducing the total cost. The idea is to use **memoisation** by storing the least cost join order for any subset that have already been computed. Therefore, we can compute them only once instead of recomputing them each the time.

To find the best join tree for a set S of n relations:

- We consider all possible plans of the form $S \bowtie (S - S_1)$ where $S \neq \emptyset$ and $S_1 \subseteq S$.
- Recursively compute the costs for joining the subsets of S to find the cost of each plan and choose the cheapest of the $2^n - 2$ alternatives
- The base case for the recursion: single relation access plan → apply all selections on r_i using the best choice of indices on r_i
- When a plan for any subset is computed, it is stored so we can reuse it again (memoisation).

This allows us to pass to a **time complexity of** $O(3^n)$ and **storage complexity** $O(2^n)$.

To get an even better result, **System R** restricts the choice to **left-deep join trees only**. The particularity of those is that the right-hand-side input for each join is a relation and not the result of an intermediate join. This reduces a lot the search space!



(a) Left-deep join tree (b) Non-left-deep join tree

For a set of n relations: consider n alternatives with one relation as right-hand side input and the other relations as left-hand side input. The **time complexity** of finding best join order is $\mathbf{O(n\ 2^n)}$ - compared to $O(3^n)$. The space complexity remains at $O(2^n)$.

3 Statistics for Cost Estimation

3.1 Role of statistics in cost-based query optimization

Statistics provide the **foundation for informed decision-making** in **cost-based query optimization**. By understanding the distribution and characteristics of the data, the **optimizer can estimate the costs** of various query execution plans and choose the most efficient path to process the query.

Here are the different variables/syntax used in this section:

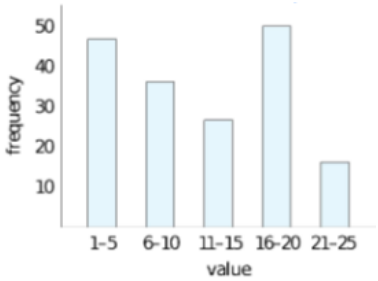
- $\boldsymbol{n_r}$ – number of tuples in a relation r
- $\boldsymbol{b_r}$ – number of blocks containing tuples of r
- $\boldsymbol{l_r}$ – size of a tuple of r
- $\boldsymbol{f_r}$ – blocking factor or r (number of tuples of r that fit into one block).
- $\boldsymbol{V(A,r)}$ – number of distinct values that appear in r for the attribute A. It is the same as the size of $\Pi_A(r)$.
- $\boldsymbol{b_r = \left\lceil \frac{n_r}{f_r} \right\rceil}$ – the number of blocks of tuples of a relation r if the tuples of r are stored together physically in a file.

Regular updates to statistics ensure that the optimizer's decisions remain accurate as the data distribution evolves.

3.2 Use of histogram for attribute statistics

Histograms allows to have **a better understanding of the distribution of the attributes of a relation**. There are two types of histograms:

- **Equi-width histograms**: you assemble the values with the **same intervals** (see picture just below), the frequency will thus vary.



- **Equi-depth histograms**: assemble the intervals to have the **same frequency** per interval. *Break up range such that each range has the same number of tuples*.

The following histogram is an equi-width histogram on attribute *age* of relation *person*.

They play a **key role in the cost-based optimization of queries**, providing the optimizer with insights into selectivity, cardinality, and skewness, ultimately leading to more efficient query execution plans.

Many **databases** also **store n most-frequent values and their counts**. A histogram is thus built on the remaining values only.

Histograms and other statistics are **usually computed based on a random sample** (not on the entire database!).

Note that when it comes to statistics, they very often become **out of date**.

- To fix this, some databases require a **analyse (vacuum)** command to be executed to update statistics.
- The others automatically recompute statistics.
    - For example, when several tuples in a relation change by some percentage.

3.3 Estimating the size of a selection and join queries

Estimating the size of a selection or join result in a relational database involves **using statistics about the underlying tables**. The key statistics that are typically used for estimation include the *cardinality* (number of rows) and *selectivity* (percentage of rows satisfying a condition) of the involved attributes.

For the notations:

- c : cost (estimated number of tuples satisfying the condition).
- r : relation
- A : column/attribute

3.3.1 Selection size estimation

$\sigma_{A=v}(r)$

➔ The cost is the **number of records that satisfy the selection criteria** (number of tuples/number of distinct values that appear in r for attribute A).

$$\rightarrow c = \frac{n_r}{V(A,r)}$$

➔ Equality condition on a key attribute (if V is a primary key)

$$\rightarrow c = 1$$

$\sigma_{A\leq v}(r)$

➔ If $v < \min(A,r) \rightarrow c = 0$

➔ If $v \geq \min(A,r) \rightarrow c = n_r$

➔ Else $c = n_r \times \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$

- If the histogram is available, then we can improve the estimate (that is for the moment just a linear interpolation).

➔ If no statistics are available $c = n_r/2$

Note that $\min(A,r)$ and $\max(A,r)$ are available in the **catalog**.

Conjonction $\sigma_{\theta_1 \wedge \ldots \wedge \theta_n}(r)$ (and)

➔ Assuming the **independence** of the conditions

➔ s_i is the number of matching tuples for the condition θ_i

➔ s_i/n_r is the **selectivity** of θ_i

$$c = n_r \frac{s_1 * s_2 \ldots * s_n}{n_r^n}$$

Disjonction $\sigma_{\theta_1 \vee \ldots \vee \theta_n}(r)$ (or)

$$c = n_r \times \left(1 - \left(1 - \frac{s_1}{n_r}\right) \times \ldots \times \left(1 - \frac{s_n}{n_r}\right)\right)$$

Negation $\sigma_{\neg\theta}(r)$

$$c = n_r - size(\sigma_\theta(r)) = 1 - \frac{s_\theta}{n_r}$$

3.3.2 Join size estimation

Cartesian product $r \times s$ contains $n_r * n_s$ tuples that each occupies $s_r + s_s$ bytes.

$$c(r \times s) = s_r + s_s$$

There are 3 different cases of joins:

- If $R \cap S = \emptyset$, then it acts as a **cartesian product**
$$c(r \bowtie s) = n_r * n_s$$
- If $R \cap S$ is a **primary key** for R, then a tuple of s will join at most one tuple from r
$$c(r \bowtie s) \leq n_s$$
- If $R \cap S$ is a **foreign key** **in** S **referencing** R, then the number of tuples in the join will be exactly the same as the number of tuples in s.
$$c(r \bowtie s) = n_s$$
- If $R \cap S = \{A\}$ is **not a key** to R or S, it is only a **common attribute that is not a key**, then we either assume that every tuple t in R produces tuples in $R \bowtie S$ or that every tuple in S produces tuples in $R \bowtie S$. We **take the minimum of the two cases**.
$$c(r \bowtie s) = \min\left(\frac{n_r * n_s}{V(A,r)}, \frac{n_r * n_s}{V(A,s)}\right)$$

3.3.3 Improvement

This is more than possible to improve those estimations thanks to the **histograms**.

Note that all statistics values are taken from the **catalog**.

3.3.4 Example

For the following relations:

```
create table student
  (ID              varchar(5),
   name                  varchar(20) not null,
   dept_name             varchar(20),
   tot_cred        numeric(3,0) check (tot_cred >= 0),
   primary key (ID),
   foreign key (dept_name) references department (dept_name)
         on delete set null
  );
```

```
create table takes
   (ID              varchar(5),
    course_id       varchar(8),
    sec_id                varchar(8),
    semester        varchar(6),
    year                  numeric(4,0),
    grade                 varchar(2),
    primary key (ID, course_id, sec_id, semester, year),
    foreign key (course_id, sec_id, semester, year) references section
(course_id, sec_id, semester, year)
          on delete cascade,
    foreign key (ID) references student (ID)
          on delete cascade
   );
```

And the following statistics (taken from the catalog):

$n_{student}$ = 5,000. $f_{student}$ = 50, which implies that $b_{student}$ =5000/50 = 100.
n_{takes} = 10000. f_{takes} = 25, which implies that b_{takes} = 10000/25 = 400.
V(ID, takes) = 2500, which implies that on average, each student who has taken a course has taken 4 courses.

- Attribute *ID* in *takes* is a foreign key referencing *student*.
- *V(ID, student)* = 5000 (*primary key!*)

Let's compute the estimate cost for the query

$$students \bowtie takes$$

As $students \cap takes$ results in ID, a **foreign key in $takes$ referencing $student$**.

$$c(students \bowtie take) = n_{takes} = 10.000$$

→ Now let's admit that there were no foreign keys to compare:

- $V(ID, student) = 5000$
- $V(ID, takes) = 2500$

$$c(students \bowtie takes) = \min\left(\frac{n_{student} * n_{takes}}{V(ID, student)}, \frac{n_{student} * n_{takes}}{V(ID, takes)}\right)$$

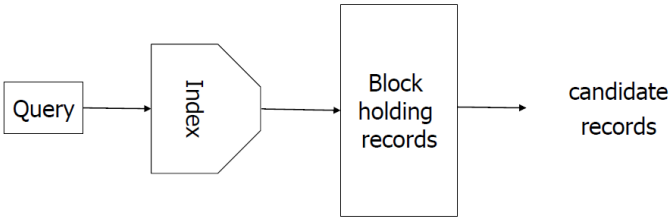
$$\rightarrow c = \min\left(\frac{5000 * 10.000}{5000}, \frac{5000 * 10.000}{2500}\right) = \min(20.000, 10.000) = 10.000$$

We can see that it is the same as if there was a foreign key!

4 Indexing

4.1 Use of indexes in query processing

An **index** is a **data structure** that provides a fast and efficient way to look up and retrieve records in a database table based on the values in one or more columns. Indexes significantly enhance the efficiency of query processing by providing **quick access paths** to specific rows **based on the values** in indexed columns.

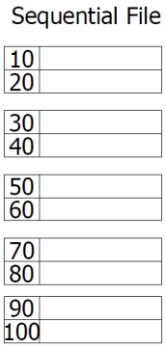


It allows to get clustered tuples at once instead of querying each tuple address in the memory. They are implemented with a specific goal each time.

Some examples in PostgreSQL: GIN, Btree, GIST, BRIN, ...

4.2 Explain these concepts

4.2.1 Sequential File



A **sequential file storage method** is the most basic method of storing data in the database. It consists of **pages** (of 4kB approximately) that are stored in blocks.

On the image on the right, we can see that the number at the beginning is the **key** to the page. It is used when accessing records, **we can't access subpages**, so we take the entire page each time.

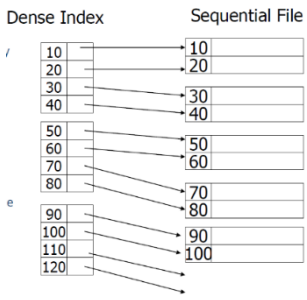
Each page has its own **unique key**. And the **sequential file** is **sorted by key value**.

4.2.2 Dense Index

A **dense index** is in the mode "**a pointer per key**". This means that one pointer points to one and only one key. This type of index does a **sequential scan of the index** to **search for a key**.

Querying a dense index is more efficient than querying a sequential file:

- It allows us to have a different order than the storage
- Thus, we can sort the data in the index based on what we are looking for.
- And searching through a sorted list is faster as we can use **binary search**.

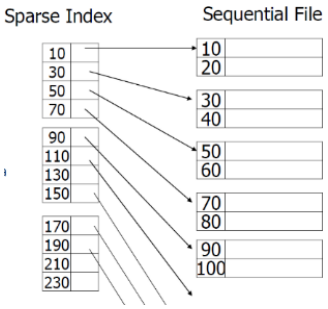


Note that it can also be used on a non-sequential file because we have a pointer that leads to a record for each record.

The **dense index** can **tell if any record exists without accessing the file** but is **bigger**, we are not always able to store the whole index in memory.

4.2.3 Sparse Index

A **sparse index** is an index of the mode **one pointer per block**.



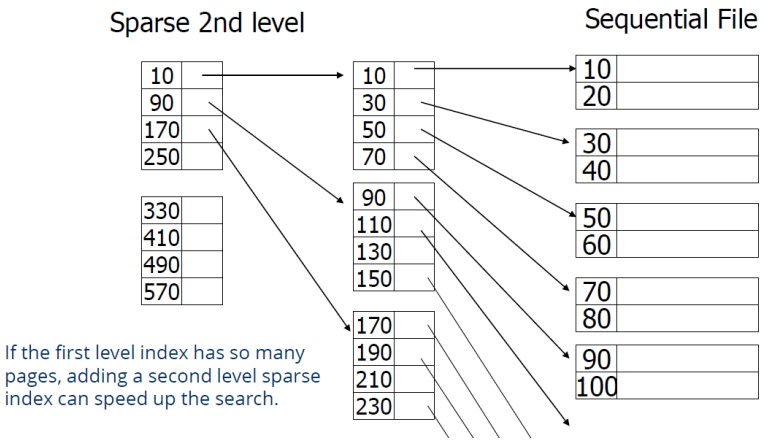
The main advantage is that there are thus **less entries in the index than in the storage**. But it is **not possible to use it on a non-sequential file** as the pages needs to be sorted to use a sparse index.

The **sparse index** has **less index space per record** and can **keep more of index in memory**. This reduces the number of access (mean cost in the database).

For example, to search for the page with the key number 25, we go to the index key 10 (because 30>25) and we go sequentially through the block pointed by the index pointer linked to the key.

4.2.4 1st and 2nd level index

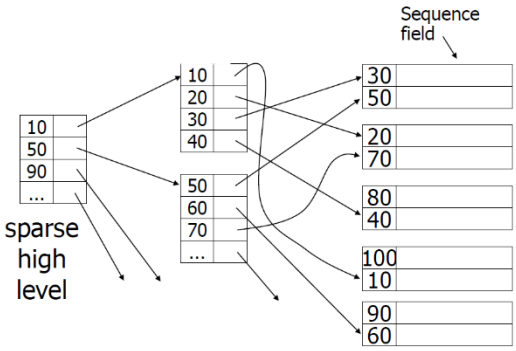
One of the main advantages of a sparse index is that it allows for **multiple levels indexes**. The example here below explains it in detail.



If the first level has too many pages, we can add a second level that will **speed up the search**.

4.2.5 Secondary indexes

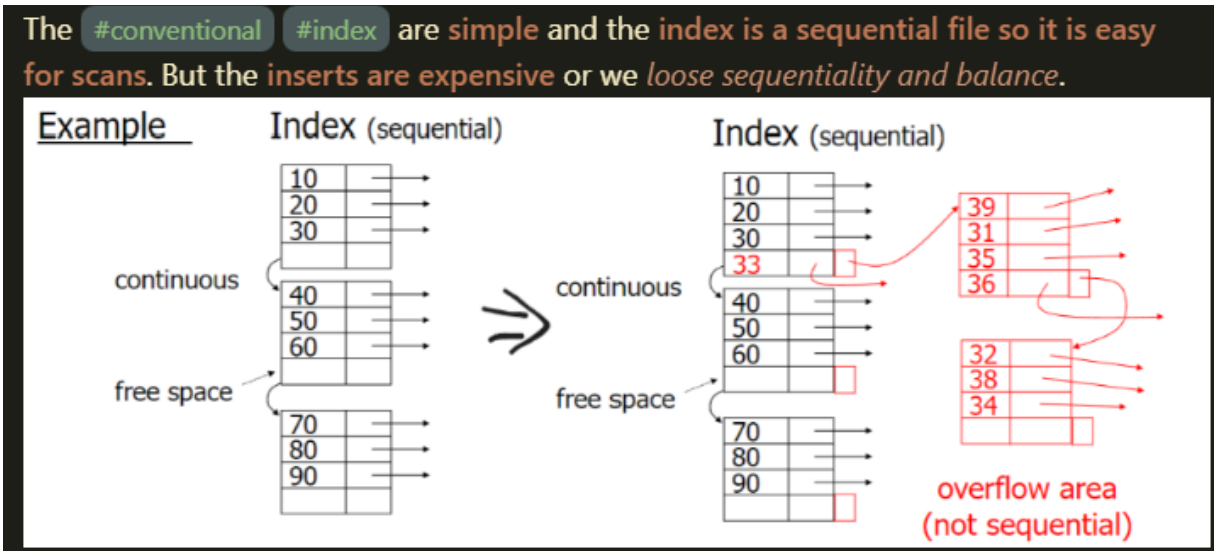
A **secondary index** is an index that applies to a **non-sequential file**. This means that the file pages **are not sorted by key**. It provides an alternative access path to the data, allowing to quickly locate and retrieve relevant rows based on the values in the indexed columns.



The **first level** cannot be a sparse index and should be a **dense index** as the pages are traditionally not sorted. The other levels can be a **sparse index**.

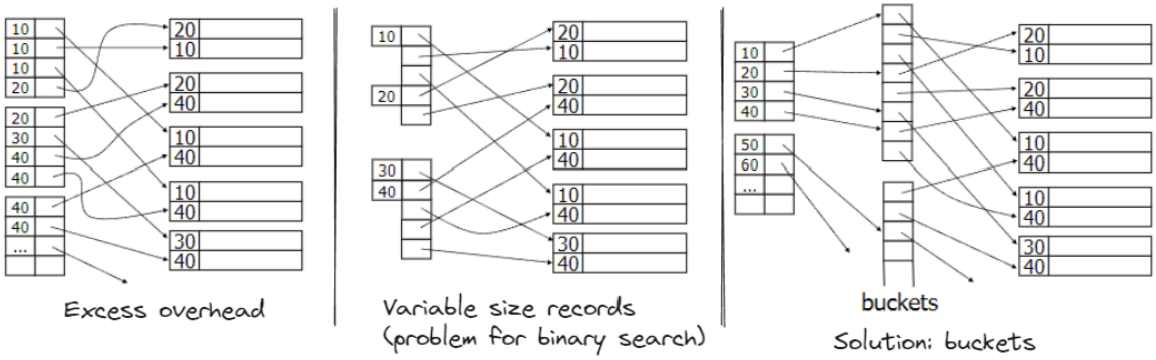
4.3 Illustrate the insertion/deletion/duplicate keys strategies in conventional indexes

This has not been done this year. But here is still a small explanation (even though it has not been seen in class).



4.4 Illustrate the benefits of buckets in secondary indexes

Something that can happen with **secondary indexes**, is that the file contains **duplicate keys**, so this causes a **problem** for the **dense index** (low level index).



The first idea is to do **excess overhead**, which consists in keeping one pointer per key. But this cannot be applied in practice as it would **take too much storage space and time**.

Another solution would be **variable size records**, one overhead for multiple pointers. But it is not possible as it would break the binary search.

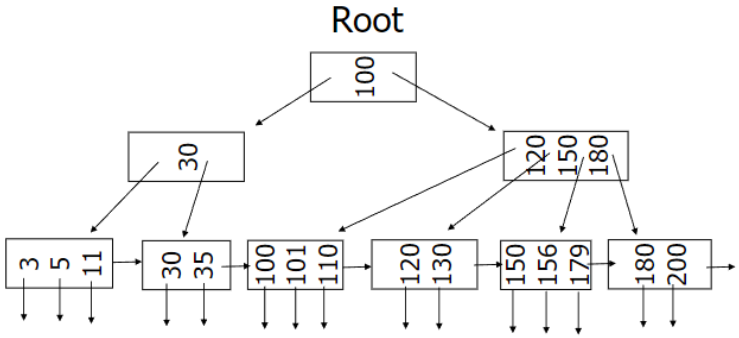
Buckets are a good solution to **avoid excess** overhead in disk space and search time while **solving the variable size** records problem in indexes. Bucket is a sort of intermediary mapping between the index and the unordered blocks, with duplicate keys file in the disk.

4.5 Illustrate the B+Tree index

https://postgrespro.com/blog/pgsql/4161516

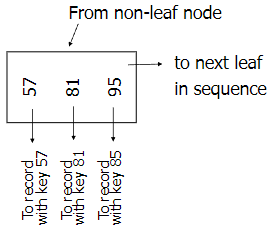
A **B+tree** (Balanced Tree) **index gives up on sequentiality** and **tries to get balance**. It is therefore a self-balancing **tree data structure** that **maintains sorted data of a column from a relation** and allows for **efficient insertion**, **deletion**, and **search operations**. It stores **all the TID** of **all rows**.

It is said to be of **order n**, with **n** the number of levels and node size. Here it is a B+tree of order 3.

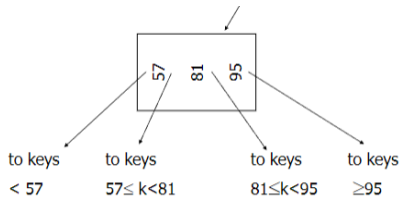


Index rows of the B+tree are **packed** into **pages/nodes**.

- In **leaf nodes**, these rows contain **data/values to be indexed** (keys) and **references to table rows**/records (**TIDs**).



- In **non-leaf nodes** (internal pages), each row **references a child node of the index and contains the minimal key value in this page**.



Each node has a **fixed size**. It has **n+1 pointers** and **n keys**. We don't want them to be empty, they should be **at least half full**:

- **non-leaf** at least $\left\lceil \frac{n+1}{2} \right\rceil$ pointers to other nodes.
- **leaf** at least $\left\lceil \frac{n+1}{2} \right\rceil$ pointers to data records in the disk.

Each node in a B+tree has the following **parameters**:

- **Keys**: The values that determine the ordering of the data.
- **Pointers to child nodes or leaf nodes**. Internal nodes have pointers to other nodes, while leaf nodes have pointers to actual data records.
- **B+trees are balanced**, that is, each leaf page is separated from the root by the same number of internal pages. Therefore, search for any value takes the same time.
- **B+trees are multi-branched**, that is, each page (usually 8 KB) contains a lot of (hundreds) TIDs. As a result, the depth of B-trees is small, up to 4–5 for very large tables.
- **Data in the index is sorted in non-decreasing order** (both between pages and inside each page), and same-level pages are connected to one another by a bidirectional list. Therefore, we can get an ordered data set just by a list walk one or the other direction without returning to the root each time.
- **The data type** that is stored must have the operators: $<, \leq, =, \geq, >$. In order to sort the values (which is mandatory for the creation of a Btree index).

Rules of a B+tree

1. All leaves at same lowest level (the tree needs to be balanced)
2. Pointers in leaves point to records except for "sequence pointer"
3. Number of pointers/keys for B-Tree

	Max ptrs	Max keys	Min ptrs→data	Min keys
Non-leaf (non-root)	n+1	n	$\lceil (n+1)/2 \rceil$	$\lceil (n+1)/2 \rceil - 1$
Leaf (non-root)	n+1	n	$\lfloor (n+1)/2 \rfloor$	$\lfloor (n+1)/2 \rfloor$
Root	n+1	n	1	1

4.6 Properties of the B+Tree to answer inequality or range searches efficiently

https://postgrespro.com/blog/pgsql/4161516

The two important properties used for inequality or range searches efficiently are:

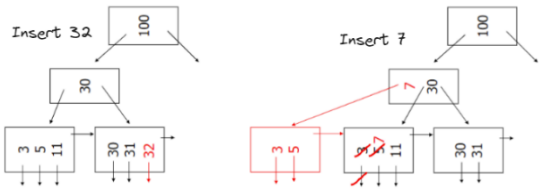
1. The **keys** stored in a Btree are **sorted**
2. The **nodes** contain **pointers** to the **previous and next node** in the same level of the tree.

We can then for example answer an inequality search by first looking for the boundary value using the equality condition and then iterate along the leaf nodes in the correct direction to return all the searched row. This is **illustrated in subsection 4.7.2.**

4.7 Illustrate the insertion, and search in B+Tree

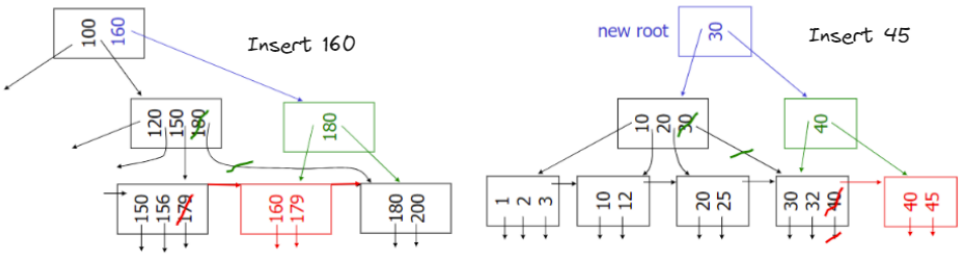
4.7.1 Insertion

Let's see some different cases for n=3:



Simple Case: Here on the left for **inserting 32**. We can see that there is already space available, therefore it is trivial.

Leaf Overflow: On the right, for **inserting 7**, we can see that we need to move the 3 and 5 to another leaf node. And because this leaf node needs to be created, we need to update the parent node too with another pointer.



Non-leaf overflow: On the left, when **inserting 160**, we need to move 179 to another leaf node. But we also need to create this new leaf node in order to assure that the tree remains balanced! Therefore there is a **non-leaf overflow** and we need to recreate a non-leaf node that will point to the new leaf node and the 180/200 leaf node.

New root: On the right, when **inserting 45**, we need to create a new leaf node and import the 40 pointer to maintain balance. This new leaf node needs to create a new parent as the previous parent can not support 4 childs as n=3. And as another non-leaf node is created, we need to create a root node.

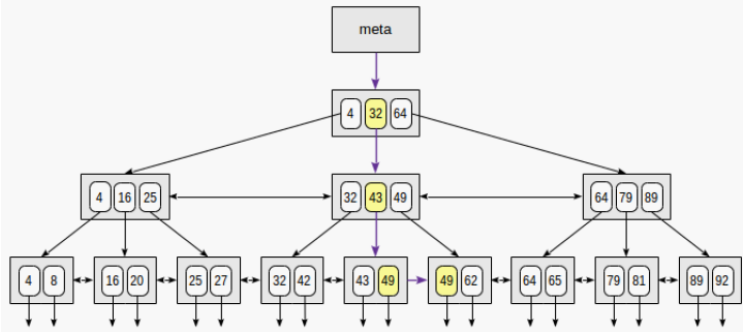
4.7.2 Search

Search by equality

Let's consider search of a value in a tree by condition **"indexed-field = expression"**. Say, we are **interested in the key of 49**.

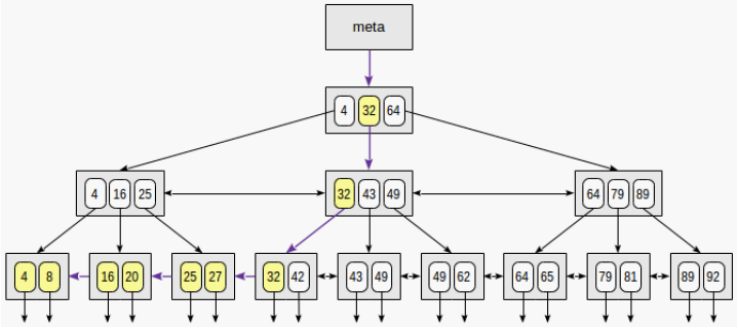
The search **starts with the root node**, and we need to **determine to which of the child nodes to descend**. Being aware of the keys in the root node (4, 32, 64), we therefore *figure out the value ranges in child nodes*. Since **32 ≤ 49 < 64**, we need to descend to the second child node. Next, the same process is **recursively repeated until we reach a leaf node from which the needed TIDs can be obtained**.

In reality, a number of particulars complicate this seemingly simple process. For example, an index can contain non-unique keys and there can be so many equal values that they do not fit one page. Getting back to our example, it seems that we should descend from the internal node over the reference to the value of 49. But, as clear from the figure, this way we will skip one of the "49" keys in the preceding leaf page. Therefore, once we've found an exactly equal key in an internal page, we must descend one position left and then look through index rows of the underlying level from left to right in search of the sought key.



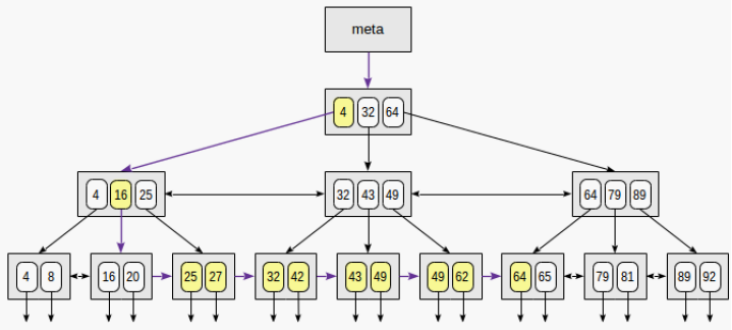
Search by inequality

When searching by the condition **"indexed-field ≤ expression"** (or "indexed-field ≥ expression"), we first find a value (if any) in the index by the equality condition "indexed-field = expression" and then walk through leaf pages in the appropriate direction to the end.



Search by range

When searching by range **"expression1 ≤ indexed-field ≤ expression2"**, we find a value by condition "indexed-field =expression1", and then keep walking through leaf pages while the condition "indexed-field ≤ expression2" is met; or vice versa: start with the second expression and walk in an opposite direction until we reach the first expression.

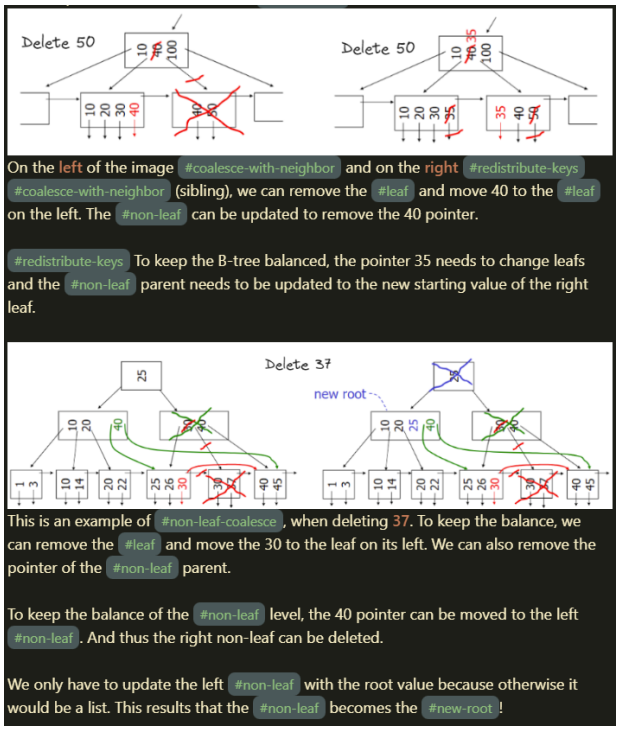


4.7.3 Deletion

This is **not for the exam**, but I did it anyway. Note that there are 4 cases too:

- Simple case
- Coalesce with neighbour
- Redistribute keys
- Non-leaf coalesce

➔ Often coalescing is not implemented as it is too hard and not worth it.



4.8 Additional information about indexes

Multi-column indexes exist and sort the values in the same order as asked. For example, for the table T(A,B), if we do

$$CREATE\ INDEX\ idx\_ab\ ON\ T(A,B);$$

The index will first sort the column A, then the column B. And thus

$$CREATE\ INDEX\ idx\_ba\ ON\ T(B,A);$$

Will sort first the column B and then A.

NULL values and their respective TIDs are stored together at the start or the end of the leaf nodes for Btree indexes.

B-tree vs B+tree indexes

In terms of **structure**:

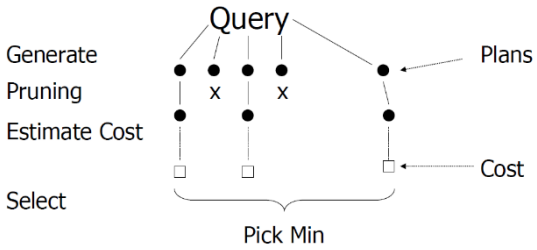
- **B-tree:** each **node contains both keys and data pointers**. Keys are not sorted, and number of keys vary within a certain range. Child pointers are associated with each key, leading to the corresponding subtrees (what we saw earlier).
- **B+tree: Keys and data pointers are separated**. All keys are present in the internal nodes, while the actual data is stored in the leaf nodes. **Leaf nodes are connected in a linked list** for efficient range queries.

For **search operations**:

- **B-tree**: can end at any level of the tree, including internal nodes. **Well suited for equality queries**.
- **B+tree**: always end at a leaf node which contains the actual data/range of data. **Well suited for range queries**.

5 Physical Query Plans

The query optimization is along these steps:



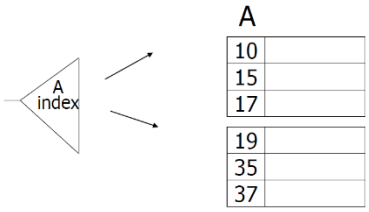
This means that we generate all plans and compare them to pick the one with the minimal cost.

To **generate the plans**, the optimiser considers: **transforming relational algebra** expression (e.g. order of joins), the **use of existing indexes**, **building indexes,** or sorting on the fly. It also uses different algorithms depending on the case, memory availability etc.

Here is some **vocabulary and syntax** for this section:

- $B(R) = \#$ blocks containing R tuples
- $f(R) = max$ # of tuples of R per block
- $M = \#$ memory blocks available
- $T(R) = \#$ tuples in the relation R (note that $T(R) > B(R)$ at all times)
- When using Btree index
  - $HT(i) = \#$ levels in index i
  - $LB(i) = \#$ leaf blocks in index i

A **clustering index** is an index that determines the **physical order of the rows** in a table. It is an **index** that allow tuples to be **read in an order** that corresponds to the **physical order**.



The **clustered file** organisation refers to the fact that blocks in the file can store tuples from multiple relations, but they are ordered.

The **clustered relation** means that B(R)<< T(R)

5.1 Different physical algorithms for joins

There are 4 different types of **join algorithms**: **nested loops**, **merge**, **with index** and **hash joins**.

$R1$ and $R2$ are tables, r, s and C are attributes (columns).

The algorithms here below are for the following transformation:

$$R1 \bowtie R2$$

5.1.1 Nested loops (iteration join)

Nested loops (iteration join): The **right relation is scanned once for every row found in the left relation**. It consists of two nested loops. For each possibility, we output the pair if it matches the join condition.

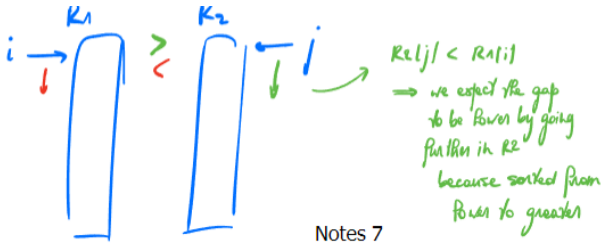
```
for each r ∈ R1 do
    for each s ∈ R2 do
        if r.C = s.C then output r,s pair
```

This strategy is **easy to implement** but can be very **time consuming**.

5.1.2 Merge join

Merge join: This join algorithm requires **both relations to be sorted initially**. Then, the **two relations** are **scanned in parallel**, and **matching rows are combined to form join rows**.

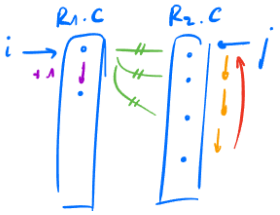
```
(1) if R1 and R2 not sorted, sort them
(2) i ← 1; j ← 1;
    While (i ≤ T(R1)) ∧ (j ≤ T(R2)) do
        if R1{ i }.C = R2{ j }.C then outputTuples
        else if R1{ i }.C > R2{ j }.C then j ← j+1
        else if R1{ i }.C < R2{ j }.C then i ← i+1
```



We can stop whenever we reach the end of R1 or R2.

The **output Tuples procedure:**

```
While (R1{ i }.C = R2{ j }.C) ∧ (i ≤ T(R1)) do
    [jj ← j;
     while (R1{ i }.C = R2{ jj }.C) ∧ (jj ≤ T(R2)) do
            [output pair R1{ i }, R2{ jj };
             jj ← jj+1 ]
    i ← i+1 ]
```



This kind of join is attractive because **each relation must be scanned only once**. The constraint resides in the fact that the relations need to be sorted, which can take some time.

To improve the merge join, we can improve by just comparing the heads while sorting. If one is lower than the others, it won't be joined anymore so we can read the next block.

5.1.3 Join with index

Join with index: This can be applied if one of the two relations has an index on the common attribute (here let's say R2.C has an index). Then for all tuples of R1, search in the index and retrieve each tuple that have the same value as r.C in C.

```
For each r ∈ R1 do                          Assume R2.C index
  [ X ← index (R2, C, r.C)
     for each s ∈ X do
          output r,s pair]
```

Note: X ← index(rel, attr, value)
then X = set of rel tuples with attr = value

This is super-efficient but needs an index.

5.1.4 Hash join

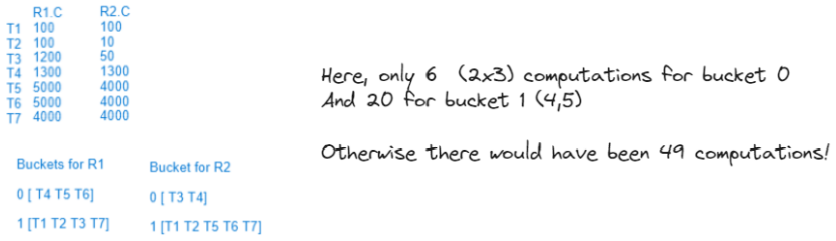
Hash join: the algorithm hashes both R1 and R2 into buckets (G for R1 and H for R2). Then the comparison can be done only between buckets with the same number (G1 with H1, G4 with H4, ...).

– Hash function h, range 0 → k
– Buckets for R1: G0, G1, ... Gk
– Buckets for R2: H0, H1, ... Hk

Algorithm
(1) Hash R1 tuples into G buckets
(2) Hash R2 tuples into H buckets
(3) For i = 0 to k do
match tuples in Gi, Hi buckets

This is faster because instead of comparing all tuples, we compare only buckets and there are less buckets than tuples.

Here is an example:



5.2 Estimate the cost of each of the four join algorithms

We must **estimate the Input/Outputs costs**, the count of the **number of disk blocks that must be read (or written) to execute query plan**.

Here imagine that we have two relations R1 and R2 that share a **common attribute** C. We know the following statistics (probably from the catalog):

- $T(R1) = 10.000$ (number of tuples in R1)

- $T(R2) = 5.000$ (number of tuples in R2)
- $S(R1) = S(R2) = 1/10$ block (size of the tuples)
- $M = 101$ (memory available)

And we want to perform the transformations

$$R1 \bowtie R2 \; or \; R2 \bowtie R1$$

5.2.1 Iteration Join (nested loops)

When the relations are **not contiguous,** is the worst case because when we read the next tuple, we must bring the next page into the memory. So, there is **1 IO/read**.

- For each tuple of R1 → $10.000 * (1 + 5000) = 50.010.000 \; IO's$
  - This is for all tuples of R1, read tuple + Read R2.
- This is way too much; therefore, we use some **memory** to make it better
  - Read 100 blocks of R1
  - Read all of R2 + join
  - Repeat until done.
  - For each R1 chunk, read chunk costs 1000 IOs and read R2 costs 5000 IOs

    → 6000 IOs

  - The computation is number of chunks of R1 * (read chunk + read whole R2)

  ⇨ $c(R1 \bowtie R2) = = \frac{10.000}{1.000} \times (1.000 + 5.000) = \mathbf{60.000} \ll 50.010.000 \; IOs$
- By using the same method, $c(R2 \bowtie R1) = \frac{5.000}{1.000} \times (1.000 + 10.000) = \mathbf{55.000} \; IOs$

Therefore, here it costs less to join R1 to R2 than the inverse. This is because there are less tuples in R2.

Now if the relations are **contiguous**, this will get better:

- Computation: for each R1 data block/memory * (read memory + Read R2)

$$c(R1 \bowtie R2) = \frac{10.000}{10*(101-1)} \times (100 + 500) = \mathbf{6.000 \; IOs}$$

- $c(R2 \bowtie R1) = \frac{5.000}{10*100} \times (100 + 1000) = \mathbf{5.500 \; IOs}$

The **general formulae for a join on contiguous unordered data is the following:**

$$c = \frac{B(R1)}{M-1}\big(M - 1 + B(R2)\big)$$

5.2.2 Merge Join

To compute the cost of the **merge join algorithm**, we need to take 3 parts into account: **sequentiality, order** and the **join**.

For **not contiguous data:**

- **Sequentiality** (read is not sequential and write is):
  - $read \; R1 + \; write \; R1 \; = \; 10.000 + 1.000 = 11.000$
  - $read \; R2 \; + \; write \; R2 \; = \; 5.000 + 500 = 5.500$
- **Order** (already sequential)
  - $2 \times \big(read(R1) + write(R1)\big) = 2 \times (1000 + 1000) = 4.000$

  - $2 \times \big(read(R2) + write(R2)\big) = 2 \times (500 + 500) = 2.000$
- **Join**: $read(R1) + read(R2) = 1.000 + 500 = 1.500$
  ⇨ $c(R1 \bowtie R2) = 11.000 + 5.500 + 4.000 + 2.000 + 1.500 = \mathbf{24.000}$ **IOs**

Now if the relations are **contiguous**, this will get better:

- **Order** (already sequential)
  - $2 \times \big(read(R1) + write(R1)\big) = 2 \times 2000 = 4.000$
  - $2 \times \big(read(R2) + write(R2)\big) = 2 \times 1000 = 2.000$
- **Join**: $read(R1) + read(R2) = 1.000 + 500 = 1.500$
  ⇨ $c(R1 \bowtie R2) = 4.000 + 2.000 + 1.500 = \mathbf{7.500}$ **IOs**

The **general formulae for a join on contiguous unordered data is the following:** (ordering using merge sort + merge).

$$c = 4 * \big(B(R1) + B(R2)\big) + B(R1) + B(R2) = 5 * \big(B(R1) + B(R2)\big)$$

5.2.3 Comparison: when to use iteration or merge.

Resolve the equation:

$$5 * \big(B(R1) + B(R2)\big) = \frac{B(R1)}{M-1}\big(M - 1 + B(R2)\big)$$

Iteration join is better than **merge join** for **contiguous unordered data** if:

$$B(R1) < 5 * \frac{B(R2)}{\frac{B(R2)}{M-1} - 4}$$

So, in our example, having $M = 101, B(R2) = 5.000$, the break point is at $B(R1) = 544$.

5.2.4 Join with index

For this algorithm, let us assume first that

- R1.C index exists and has 2 levels
- R2 is **contiguous and unordered**
- R1.C index fits in memory

The last assumption tells us that querying the index is free, the only costs will come from the reading of all tuples of R2. This assumption is not always valid as it is really restrictive.

The cost will follow this "pseudo-algorithm":

- Read 5.000/10=500 blocks
- For each R2 tuple
  - Probe index
  - If match, read R1 tuple → 1 IO (indeed, outputting a pair requires to read a block of R1).

Therefore, if:

1. **R1.C** is a **key** and **R2.C** is a **foreign key**, then **maximum 1 matching value** since keys are unique

➔ **Total cost =** $500 + 5.000 * (1) * 1 = 5.500\ IOs$

2. Say V(R1, C)= 5.000 and T(R1)=10.000 with **uniform assumptions**, then expect 10.000/5.000 = 2

➔ **Total cost =** $500 + 5.000 * (2) * 1 = 10.500\ IOs$

3. Say that the domain of C DOM(R1,C)[1] = 1.000.000 and T(R1) = 10.000, with **alternate assumption**, then expect 10.000/1.000.000=1/100% of matching tuples.

➔ **Total cost =** $500 + 5.000 * (1/100) * 1 = 550\ IOs$

→ One semi-general rule could be: $c = B(R2) + \frac{T(R1)}{V(R1,C)} * T(R2)$

This requires one more parameter, the **selectivity**.

Now let's see what happens **if the index does not fit in memory**. For example, R1.C index is 201 blocks and memory is 101 blocks.

- Then we keep the root + 99 leaf nodes in memory
- The expected cost of each probe is (probability of finding it + probability of needing to search for it in memory), the cost for probes is in parenthesis.

$$E = (0) * \frac{99}{200} + (1) * \frac{101}{200} \approx 0.5$$

The **total cost, including probes becomes**:

2. 500+5.000 (Probe + get records) = 500+5.000* (0.5+2) = 500+12.500 = **13.000 IOs**
3. 500+5.000 (0.5 * 1 + 1/100 * 1) = 500+2.500+50 = **3050 IOs**

5.2.5 Hash Join

If we **assume that R1 and R2 are contiguous and unordered**, that we use **100 buckets**.

First, we **bucketize:** read R1, hash + write in buckets, and that the same is done for R2.

➔ Read R1 + write, Read R2 + write

Then we do the following for all buckets:

- Read one R1 bucket, built the memory hash table
- Read corresponding R2 bucket + hash probe in the memory hash table

➔ Read R1, R2

Total Cost = $3 \times (1.000 + 500) = \mathbf{4.500\ IOs}$

not contiguous	Iterate R2 ⋈ R1	55,000 (best)
	Merge Join	_____
	Sort+ Merge Join	_____
	R1.C Index	_____
	R2.C Index	_____
contiguous	Iterate R2 ⋈ R1	5500
	Merge join	1500
	Sort+Merge Join	7500 → 4500
	R1.C Index	5500 → 3050 → 550
	R2.C Index	_____

[1] The domain of an attribute is its possible range.

5.3 Illustrate the memory requirement of the merge and hash join algorithms

5.3.1 Merge join

The memory requirement for **merge sorts** depends on the number of chunks. Let's say that we have $\boldsymbol{k}$ **blocks in memory** and $\boldsymbol{x}$ **blocks for relation sort**.

$$\#chunks = x/k \ \ and\ te\ size\ of\ a\ chunk\ k$$

$$\#chunks \leq buffers\ available\ for\ merge$$

So $\frac{x}{k} \leq k => \boldsymbol{k \geq \sqrt{x}}$

⇨ The **number of pages available in memory** must be **higher than the square root of the number of blocks in the relation**.

So, for our example, R1 = 1.000 blocks thus → $k \geq 31.62$ ➔ 32 blocks of memory are needed.

Note that R2 is smaller thus we **compute for the most restrictive relation** (R1 here).

5.3.2 Hash join

The memory requirements for the **hash join** depends on the **size of the buckets**.

The size of the bucket (of R1) x/k

- X = number of R1 blocks
- K = number of memory buffers

➔ $\frac{x}{k} < k$ → $\boldsymbol{k > \sqrt{x}}$

Therefore, we need k+1 total memory buffer.

6 Extending database systems

https://www.postgresql.org/files/documentation/pdf/16/postgresql-16-A4.pdf

6.1 Architectural components that make PostgreSQL extensible

PostgreSQL is extensible because **its operation is catalog-driven**. If you are familiar with standard relational database systems, you know that they store information about databases, tables, columns, etc., in what are commonly known as system catalogues. (Some systems call this the data dictionary.) The catalogues appear to the user as tables like any other, but the DBMS stores its internal bookkeeping in them. One key difference between PostgreSQL and standard relational database systems is that PostgreSQL stores much more information in its catalogues: not only information about tables and columns, but also information about data types, functions, access methods, and so on. These tables can be modified by the user, and since PostgreSQL bases its operation on these tables, this means that PostgreSQL can be extended by users. By comparison, conventional database systems can only be extended by changing hardcoded procedures in the source code or by loading modules specially written by the DBMS vendor. (cf. manual PostgreSQL 16)

6.1.1 What is the role of the catalog ?

The **system catalog** is the place where a relational database management system stores **schema metadata**, such as information about tables and columns, and internal bookkeeping information. PostgreSQL's system catalogues are regular tables. You can drop and recreate the tables, add columns, insert, and update values, and severely mess up your system that way. Normally, one should not change the system catalogues by hand, there are normally SQL commands to do that. (cf. manual PostgreSQL 16)

6.1.2 How is PostgreSQL able to process user types ?

PostgreSQL provides a framework for users to define and process user-defined types with **various aspects such as storage, input/output functions, and statistics**.

Storage

Users can **define their own composite types** using the CREATE TYPE command. A composite type consists of **multiple attributes, each with its own data type**. During type creation, users can **specify how the data should be stored internally**. This includes considerations for alignment, padding, and the storage format of each attribute.

Input/output

Users can define **input functions** (also known as "casting functions") that **convert an external representation of a type into the internal storage format**. Input functions are defined using the CREATE FUNCTION command with the AS clause specifying the transformation logic.

Users can define **output functions** that **convert the internal storage format back into an external representation**. Output functions are also defined using the CREATE FUNCTION command.

Statistics

Users can define **statistics functions** for their types, **providing information about the distribution of data**. Statistics functions are **useful for query optimization** by providing the query planner with insights into the characteristics of the data.

6.1.3 How is PostgreSQL able to compute functions over user types ?

Users can define functions using the CREATE FUNCTION SQL command. The function definition includes the function name, input parameters (including user-defined types), return type, and the function body.

```
CREATE FUNCTION mytype_function(mytype) RETURNS int
AS 'SELECT ...' LANGUAGE SQL;
```

6.1.4 How is PostgreSQL able to use its generalized index structures over user types ?

PostgreSQL supports **user-defined index methods that can be associated with user-defined types**. Users can create index methods tailored to the characteristics of their types.

```
CREATE INDEX mytype_index ON mytable USING mytype_ops (mycolumn);
```

6.1.5 What is the role of extensions in PostgreSQL ?

Extensions simplify the management and installation of additional functionalities beyond the core PostgreSQL distribution. Moreover, they **contribute to process of adding new features**, managing dependencies, and adapting the database to diverse application requirements.

6.2 Steps to create a PostgreSQL extension

To define an extension, you need **at least a script file that contains the SQL commands to create the extension's objects**, and a **control file** that specifies a few basic properties of the extension itself. If the extension includes C code, there will typically also be a shared library file into which the C code has been built. Once you have these files, a simple CREATE EXTENSION command loads the objects into your database.

The CREATE EXTENSION command relies on a control file for each extension, which must be named the same as the extension with a suffix of .control, and must be placed in the installation's SHAREDIR/extension directory. There must also be at least one **SQL script file, which follows the naming pattern extension--version.sql** (for example, foo--1.0.sql for version 1.0 of extension foo). By default, the script file(s) are also placed in the SHAREDIR/extension directory; but the control file can specify a different directory for the script file(s). (cf. manual PostgreSQL 16)

complex.control : This file is a control file that provides metadata about the extension, including its name, version, and other properties.

complex--1.0.sql : This SQL file contains the SQL statements necessary to install the extension, such as creating tables, functions, or other database objects required by the extension.

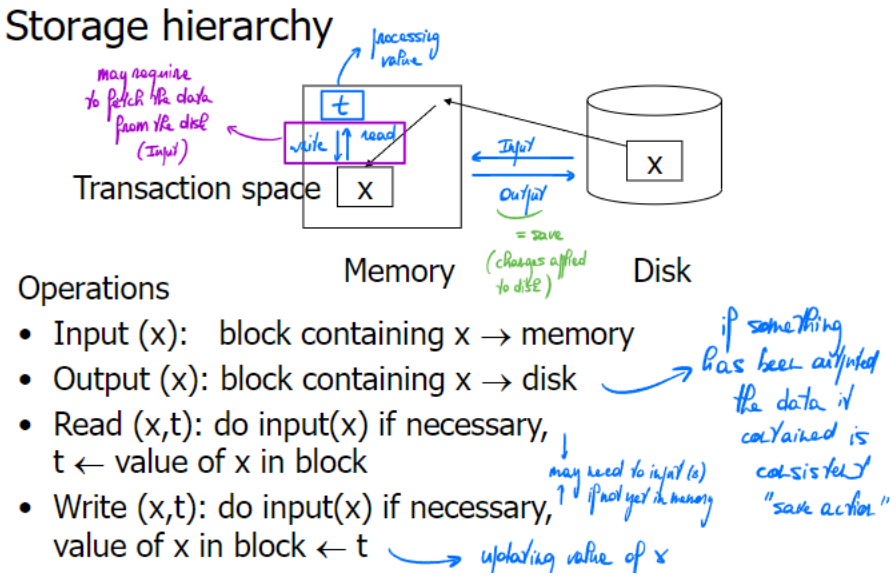
complex.c : This is the C source code file that implements the functions and behaviours of the extension. It contains the actual code for the extension's functionality.

When you install the extension in PostgreSQL, the control file (complex.control) is read to obtain metadata about the extension. The SQL file (complex 1.0.sql) is executed to create the required

database objects, and the C file (complex.c) provides the actual implementation of the extension's functions.

7 Failure Recovery

The storage hierarchy and operations considered are described here below:



When something is in the memory, it is not considered safe. When it is on the disk (database), it is safe (and thus must agree to the constraints).

7.1 Concept of a database transaction

7.1.1 Consistency and constraints

The predicates data must always satisfy the **integrity** or **consistency constraints**. Here are some examples:

- x is key of relation R
- $x \rightarrow y$ holds in R
- Domain(x) = {Red, Green, Blue}
- α is valid index for attribute x of R
- no employee should make more than twice the average salary
- …

A **state** is said to be **consistent** if **it satisfies all constraints**.

A **database** is said to be **consistent** if **it is in a consistent state**.

It is important to know that a database cannot always be in a consistent state, for example there are transitions states that it must pass by. Therefore, we use the concept of **database transaction**.

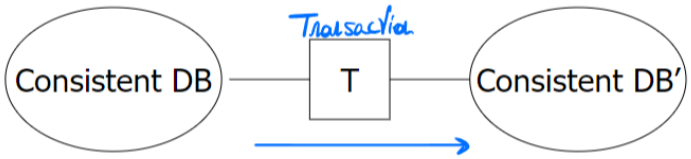
7.1.2 Database transaction

A **database transaction** is a **unit of work that consists of one or more operations** performed on a database. These operations can include inserting, updating, deleting, or querying records within one or more database tables.

BEGIN TRANSACTION;

```
UPDATE Account SET Balance = Balance - 100 WHERE AccountID = 123;
UPDATE Account SET Balance = Balance + 100 WHERE AccountID = 456;
COMMIT;
```

The most important **property** of a **transaction** is that it **preserves the consistency** of the database. It can be seen as a vector of transition between two consistent states of the database.



The advantage of this is that we can always revert the transaction to come back to an older consistent state thanks to the following **assumption**:

if T start in a consistent state + T executes in isolation → T leaves the consistent state

Note that a **transaction will not make the database consistent** if it was not earlier (or with a lot of luck).

If the database crashes during a transaction, it loses its consistency !

7.2 Purpose crash recovery methods and logging

Assume that we have the **constraint A = B** for the database and that we want to perform the following operations (in this order):

- Read(A,t); t ← t * 2; write(A,t);
- Read(B,t); t ← t * 2; write(B,t);
- Output(A); Output(B);

For the list of operation above, it will **intuitively** work that way:

- Read the value of A in disk to put it in the memory, apply multiplication (x 2), write it on the memory and write it on the disk/database.
- Read the value of B in disk to put it in the memory, apply multiplication (x2), write it on the memory and write it on the disk/database.

This is ok, but **if a crash occurs** between Output(A) and Output(B). Then A will be modified, but not B. Thus, the constraint will not be respected, leading to an **inconsistent state**.

7.3 Illustrate undo logging, and the associated crash recovery

The **undo logging** is a method to assure the database to arrive in a consistent state even if a crash occurs during a transaction. The principle consists of **logging important events** to keep track of the changes in case of system failure. This principle is also called **immediate modification**. The main **problem** is that it *increases the IO costs* as we put data to the disk early and free the memory and we flush more often.

- Read(A,t); t ← t * 2; write(A,t);
- Read(B,t); t ← t * 2; write(B,t);
- Output(A); Output(B);

The **undo logging** will do the following:

- **Write <T1, start> in the log**

- Read the value of A in disk to put it in the memory, **log <T1, A, 8>**, apply multiplication (x 2), write it on the memory and write it on the disk.
- Read the value of B in the disk to put it in the memory, **log <T1, B, 8>**, apply multiplication (x2), write it on the memory and write it on the disk.
- **Write <T1, commit> in the log**

The change from the original method is that we write in the log at each important event so that we know which values can be problematic if a crash occurs.

There still can be **complications** the **log is written in the memory first and not on the disk on every actions**. Therefore, if a crash occurs before that the log is written in the disk, there will be some problems. This depends on the **log manager**.

Undo logging protocol

1. For every action generate undo log record (containing old value)
2. Before x is modified on disk, log records pertaining to x must be on disk (write ahead logging: **WAL**)
3. Before commit is flushed to log, all writes of transaction must be reflected on disk

Example

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<T, A, 8>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T, B, 8>
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)	OUTPUT(B)	16	16	16	16	16	
11)							<COMMIT T>
12)	FLUSH LOG						

Crash recovery for undo logging

To recover the consistency in case of a system failure, the following rules needs to be followed:

(1) Let S = set of transactions with
<Ti, start> in log, but no
<Ti, commit> (or <Ti, abort>) record in log

(2) For each <Ti, X, v> in log,
in reverse order (latest → earliest) do:
- if Ti ∈ S then
  - write (X, v)
  - output (X)

(3) For each Ti ∈ S do
- write <Ti, abort> to log

It is important to notice that we need to run through the log in reverse order (latest to earliest/bottom up).

Example: If a **failure** happens **between Output(A) and Output(B)** (9/10), we start and see that the previous values are logged:

- Undo A = 16 → 8
- Undo B = 8 → 8

It is important not to commit.

If a **failure happens during recovery,** it is **not a problem** as **undo is idempotent** (doing it once or 2384723 times is the same).

7.4 Illustrate redo logging, and the associated crash recovery

In **redo logging**, the idea is to **log on disk before the modification** (compared to undo logging where we first made the changes then logged it!). The main **problem** is that we **need to keep all modified blocks in memory** until commit, but there are less IOs as the values stays in the memory longer.

- Read(A,t); t ← t * 2; write(A,t);
- Read(B,t); t ← t * 2; write(B,t);
- Output(A); Output(B);

For these instructions, the **redo logging** will compute it this way:

- Read the value of A in disk to put it in the memory, apply multiplication (x 2), write it on the memory.
- Read the value of B in the disk to put it in the memory, apply multiplication (x2) and write it on the memory.
- **Write <T1, start> in the log**
- **log <T1, A, 16>**
- **log <T1, B, 16>**
- **Write <T1, commit> in the log**
- Write new values of A and B on the disk.
- **Write <T1, end> in the log**

As we can see, the value is first changed in memory, then the logs are written, then the output is changed in the disk and finally a logging is made to ensure that it is finished.

Redo logging protocol

1. For every action, generate redo log record (containing new value)
2. Before X is modified on disk (DB), all log records for transaction that modified X (including commit) must be on disk
3. Flush log at commit
4. Write END record after DB updates flushed to disk

Example

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<$T, A, 16$>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<$T, B, 16$>
8)							<COMMIT T>
9)	FLUSH LOG						
10)	OUTPUT(A)	16	16	16	16	8	
11)	OUTPUT(B)	16	16	16	16	16	

Crash recovery for redo logging

To recover the consistency in case of a system failure, the following rules needs to be followed:

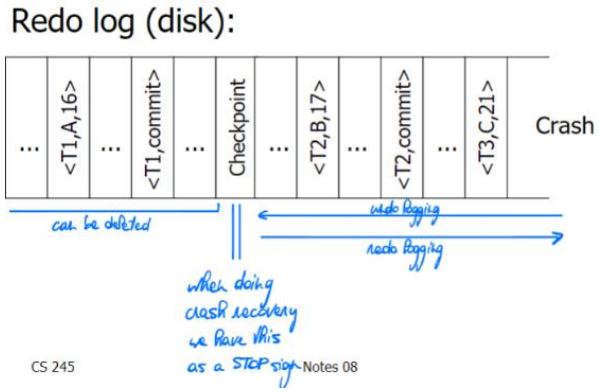
(1) Let S = set of transactions with
<Ti, commit> (and no <Ti, end>) in log
(2) For each <Ti, X, v> in log, in forward
order (earliest → latest) do:
- if Ti ∈ S then { Write(X, v); Output(X) }
Top down
(3) For each Ti ∈ S, write <Ti, end>

This time, the order is top-down (earliest to latest). We are not returning in previous state, but we **finish the transactions that are not finished**. And they appear in order in which they were requested.

7.5 Describe the concept and benefit of checkpoints

The main advantage of **checkpoints** is that it allows to **delay DB flushes** when we have several transactions that modifies the same data as it **is expensive**.

Checkpoints consists in **marking the log** such that **all that came before does not need to be reprocessed** in case of crash recovery. It is good as it allows to **not have an ever-increasing log** that we need to scan entirely at each crash.



Instead of writing <Ti, end> actions on the log, write simple checkpoints.

This is done **periodically**:

1. Do not accept new transactions → freeze the DB periodically, blocking new transactions
2. Wait until all transactions finish
3. Flush all log records to disk (log)
4. Flush all buffers to disk (DB) (do not discard buffers)
5. Write "checkpoint" record on disk (log)
6. Resume transaction processing

7.6 Illustrate undo/redo logging, and the associated crash recovery

Undo/Redo logging allows to solve the problems of **undo and redo logging** that have too many IOs or too many blocks stored in memory.

An update in the database will be logged by using the following log message:

<Ti, Xid, New X val, Old X val>

Where Xid is the reference to the page.

Undo/Redo logging rules

1. Page X can be flushed before or after Ti commit
2. Log record flushed before corresponding updated page (**WAL** – write ahead logging)
3. Flush at commit (log only)

The idea is to log before we output (**WAL**).

Undo/Redo example

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<$T, A, 8, 16$>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<$T, B, 8, 16$>
8)	FLUSH LOG → WAL						
9)	OUTPUT(A)	16	16	16	16	8	
10)							<COMMIT T>
11)	OUTPUT(B)	16	16	16	16	16	↳ no need to flush

Crash recovery procedure

If a commit is there, apply **redo relogging** to ensure that committed changes which may be in the buffer but not yet written in the disk can be recovered by reapplying the changes from the redo log.

If there is **no commit**, it implies that the changes made by the transaction are not considered permanent. So, we apply **undo logging** to roll back the incomplete transaction and maintain the consistency of the database. Note that **simple checkpoints are not affordable anymore**. The logging size becomes critical, and we have to scan the whole log.

8 Concurrency control

8.1 Explain how concurrent transactions can lead to violation of consistency

The principle of **concurrent transactions** is that **multiple transactions** (for example coming from different users) **access the database at the same time**.

This can lead to a **violation of the consistency** when the **same page is reached/modified by several concurrent transactions**. This depends on the **schedule** of the transactions.

For this section, let us assume that we have a **constraint** A=B and two **transactions** T1 and T2 that are described as follow:

T1:	T2:
Read(A)	Read(A)
A ← A+100	A ← A×2
Write(A)	Write(A)
Read(B)	Read(B)
B ← B+100	B ← B×2
Write(B)	Write(B)

Schedule A

T1	T2	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
Read(B); B ← B+100;			
Write(B);			125
	Read(A);A ← A×2;		
	Write(A);	250	
	Read(B);B ← B×2;		
	Write(B);		250
		250	250

Schedule B

T1	T2	A	B
		25	25
	Read(A);A ← A×2;		
	Write(A);	50	
	Read(B);B ← B×2;		
	Write(B);		50
Read(A); A ← A+100			
Write(A);		150	
Read(B); B ← B+100;			
Write(B);			150
		150	150

Schedule C

T1	T2	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
	Read(A);A ← A×2;		
	Write(A);	250	
Read(B); B ← B+100;			
Write(B);			125
	Read(B);B ← B×2;		
	Write(B);		250
		250	250

Schedule D

T1	T2	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
	Read(A);A ← A×2;		
	Write(A);	250	
	Read(B);B ← B×2;		
	Write(B);		50
Read(B); B ← B+100;			
Write(B);			150
		250	150

Here, the consistency is not respected!

As we can see with schedule D, we lose the consistency of the database, schedule C succeeds but this is pure luck!

In order to understand this chapter, here are some definitions:

- A **schedule** represents the chronological order in which actions will be processed.
- An **action** is either a read, write, etc

- $r_i(A)$ – Transaction i reads the value[2] A
- $w_i(A)$ – Transaction i writes the value A

- A **transaction** is a sequence of actions.

8.2 Describe the concepts of serializable schedule and conflict-serializable schedules

A **serializable schedule** is a schedule where it is **possible to perform transitions between actions to have a serial procedure**.

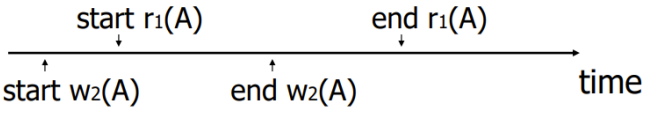
A **serial schedule** is a schedule where there are **no interleaving of actions or transactions**, each transaction is executed in its entirety before the next one begins.

Several actions are said to be **conflicting actions** if there is **at least one of the operations is a write** on the same data item as another action.

Two schedules are said to be **conflict equivalent** if one of them **can be transformed into the other by a series of swap of non-conflicting actions**.

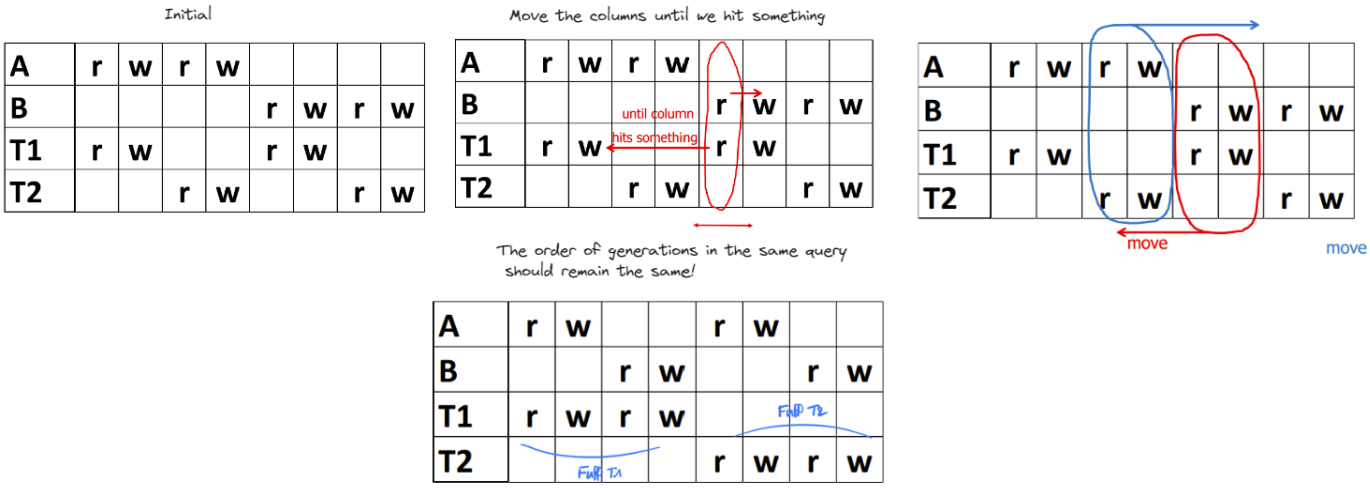
We say that a schedule is **conflict serializable** when it is **conflict equivalent to some serial schedule**.

For example, here it is equivalent to do $r_1(A)w_2(A)$ or $w_2(A)r_1(A)$.



This assumption is called **atomic actions** and is a low-level synchronisation mechanism.

To shuffle the steps of the procedure while maintaining the data consistency, we can play the **transaction game**.



When we cannot move anymore, if the schedule is serialised, it is ok. But if not, is said to be non-serializable.

[2] The term « value » is poorly chosen here

8.3 Illustrate the use of a precedence graph for checking conflict serializability

Instead of trying to win the **transaction game** for all schedules. The **precedence graph for a schedule S is noted P(S)** allows us to **determine whether a schedule is serializable** or not.

To build the precedence graph, we need to do the following:

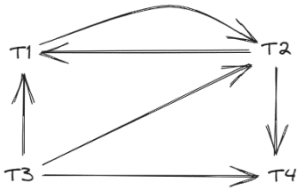
- Each **transaction** is represented by a **node**
- We write an **arc** from Ti to Tj whenever:
  - $p_i(A), q_j(A)$ are **actions** in S
  - $p_i(A) <_S q_j(A)$ (this means that p is before in the schedule S)
  - At least one of p_i, q_j is a write.

We know that the **schedule** is **serializable** if we **do not have a cycle** in the precedence graph.

Here is an example for the schedule:

$$S = w_3(A)\ w_2(C)\ r_1(A)\ w_1(B)\ r_1(C)\ w_2(A)\ r_4(A)\ w_4(D)$$

- There are arcs from T3 to all Ti because $w_3(A)$ comes before all actions on A from all Ti
- There is an arc from T2 to T1 because $w_2(C)$ comes before the $r_1(C)$
- There is an arc from T2 to T4 because $w_2(A)$ comes before the $r_4(A)$
- There is an arc from T1 to T2 because $r_1(A)$ comes before $w_2(A)$



8.4 Verify whether a schedule is well-formed, legal, and implements 2PL

Another way to determine if a schedule is serializable is to use a **scheduler**, but in order to use it, we first need to define two new actions:

- Exclusive **lock** $l_i(A)$ – lock the access to the data A for the transaction i
- **Unlock** $u_i(A)$ – unlock the access to the data A for the transaction i

These actions will be used to form a **lock table** will **help the scheduler** to check whether a schedule is serializable or not.

The scheduler will check whether the schedule respects the **3 rules**: **well-formed transactions**, **legal scheduler,** and **two-phase locking for transactions (2PL)**.

8.4.1 Rule 1 – Well-formed transactions

All transactions in the schedule **must maintain**:

- **Consistency** of the DB. The locks acquired and released during a transaction should be such that the overall integrity and consistency of the DB are preserved.
- **Isolation** – concurrent transactions do not interfere with each other by acquiring conflicting locks
- **Atomicity** – each transaction is treated as a single indivisible unit of work.

- If a transaction acquires locks on multiple data items, it should release all those locks together when the transaction is committed or roll back all the locks if the transaction is aborted.

8.4.2 Rule 2 – Legal scheduler

A transaction cannot lock an item while another transaction already locked the data. It has to wait for the data to be unlocked in order to access it.

$$S = \ldots\ldots\ l_i(A) \underset{\text{no } l_j(A)}{\longleftrightarrow} \ldots\ldots\ u_i(A) \ldots\ldots$$

For example, here let's fix this schedule:

$$S1 = l_1(A)\ l_1(B)\ r_1(A)\ w_1(B)\ l_2(B)\ u_1(A)\ u_1(B)\ r_2(B)\ w_2(B)\ l_3(B)\ r_3(B)\ u_3(B)$$

Here we can see that T2 locks the data B even though it is already locked by T1.

$$S2 = l_1(A)\ l_1(B)\ r_1(A)\ w_1(B)\ u_1(B)\ l_2(B)\ u_1(A)\ u_1(B)\ r_2(B)\ w_2(B)\ l_3(B)\ r_3(B)\ u_3(B)$$

When we add the unlock for T1 of data B before the lock, it is correct. But then we have a problem that at the end T2 never unlocks B.

$$S3 = l_1(A)\ l_1(B)\ r_1(A)\ w_1(B)\ u_1(B)\ l_2(B)\ u_1(A)\ u_1(B)\ r_2(B)\ w_2(B)\ u_2(B)\ l_3(B)\ r_3(B)\ u_3(B)$$

This last schedule is correct.

8.4.3 Rule 3 – Two phase locking (2PL)

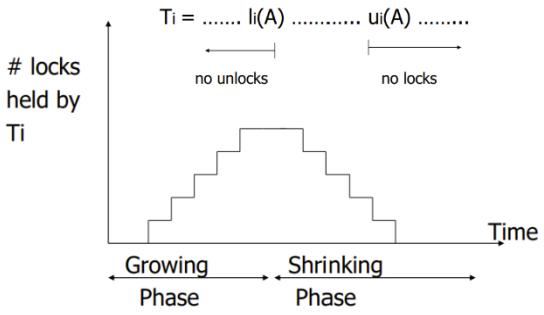
Let's find some motivation that these two rules are not sufficient.

We can see that the schedule on the right does not maintain consistency. Even though it respects the two first rules.

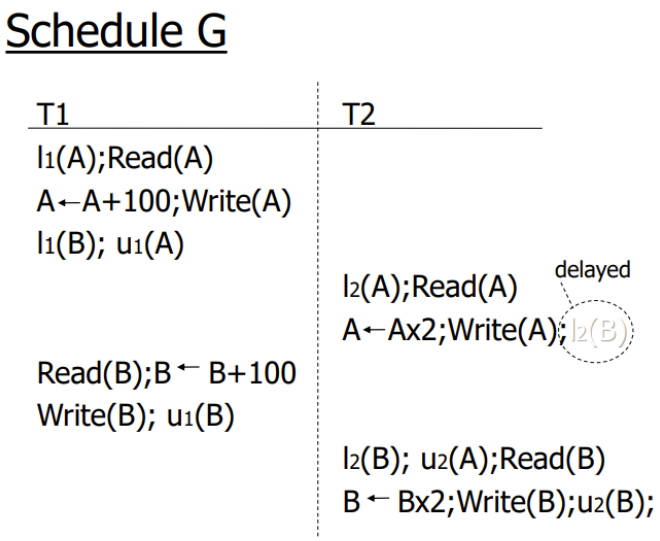
T1	T2	A	B
		25	25
l_1(A);Read(A)			
A←A+100;Write(A);u_1(A)		125	
	l_2(A);Read(A)		
	A←Ax2;Write(A);u_2(A)	250	
	l_2(B);Read(B)		
	B←Bx2;Write(B);u_2(B)		50
l_1(B);Read(B)			
B←B+100;Write(B);u_1(B)			150
		250	150

The last rules consist in **two phase locking** for transactions and ensures that at any point in time during the execution of the transaction, **the system is in a state that could arise if transaction were executed in some serial order**.

1. **Growing phase** (locking phase) – A transaction is allowed to acquire as many locks as needed but cannot release any locks.
2. **Shrinking phase** (unlocking phase) – This phase is entered as soon as a transaction release its first lock. The transaction is not allowed to acquire any new locks but can release the locks it currently holds



An example of application is for the following schedule, as we can see the $l_2(B)$ needs to wait for the unlock of it by T1. There is thus a delay.



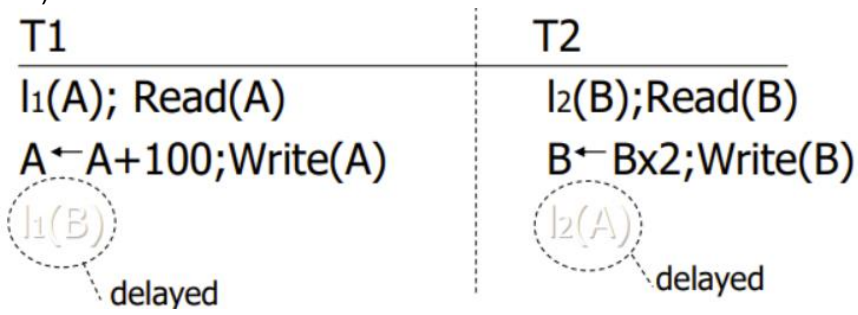
8.5 Illustrate the concurrency issues that can happen when the three rules are not implemented

There are several examples, one is already shown in the beginning of Section 8.4.3.

8.6 Explain increment locks, update locks, shared locks, and multi-granular locks

No need to memorize the compatibility matrices. They will be given if needed.

As we will see in future examples, **2PL is a subset of all serializable schedules**. This means that it is **too restrictive**. Therefore, there can be problems with 2PL schedules such as **deadlocks** (see example below).



8.6.1 Shared locks

Therefore, **shared locks** become interesting. It modifies slightly the locks to have two different locks/unlocks possible

- X for **exclusive locks** – usually used when a write will be done.
- S for **shared locks** – usually used for reads.

Those are noted $l_{Si}(A)$ or $l_{Xi}(A)$.

The 3 rules are thus adapted for it.

1. **Well-formed transactions** – The transaction that reads and writes the same object have two options
    - Either request an exclusive lock.

$$T1 = l_{x_1}(A) \dots r_1(A) \dots w_1(A) \dots u_1(A) \dots$$

- Or upgrade a shared lock to an exclusive lock.

$$T1 = l_{S_1}(A) \dots r_1(A) \dots l_{X_1}(A) \dots w_1(A) \dots u_1(A) \dots$$

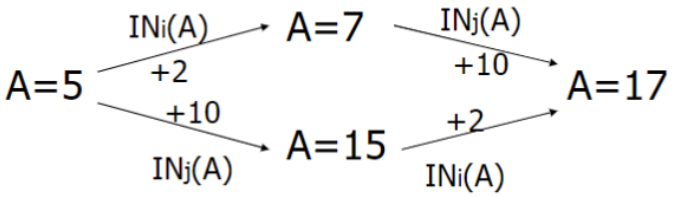
2. **Legal scheduler** – We talk about a **compatibility matrix** that determines whether we can or not lock with the following type or not for different transactions.
   The first column represents the first transaction that already locked, and the first row is the other transaction that wants to know whether it can lock or not.
   We can see that **shared-locks** can coexist with other **shared-locks**.

	S	X
S	true	false
X	false	false

3. **2PL transactions** – Not a lot of changes except for upgrades:
   - If upgrade gets more locks, → then no change
   - If upgrade releases read → shared lock can be allowed in growing phase.

8.6.2 Increment locks

Increment locks are used to **control access to increment operations on numerical values in a database**. When **multiple transactions attempt to increment the same value simultaneously**, conflicts can arise, leading to incorrect results and potential data inconsistencies. But here let us assume that we have cases where those do not conflict (see example below).



This is represented by **atomic increment action**:

$$IN_i(A) = \{read(A); A \leftarrow A + k; write(A)\}$$

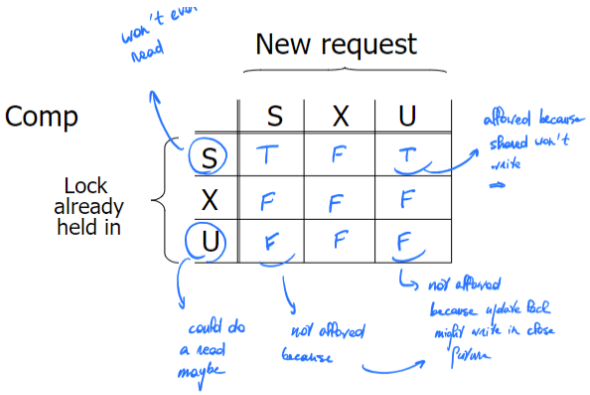
	S	X	I
S	T	F	F
X	F	F	F
I	F	F	T

The **compatibility matrix** is the following.

8.6.3 Update locks

Update locks are **used to fix** the following **deadlock** problem: "when two transactions share locks the same data and wants to update their locks to an exclusive lock". This would not respect the 2PL transaction rule.

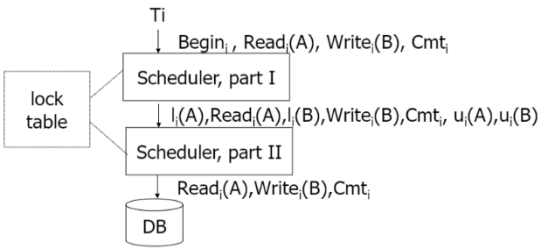
Instead of requesting a shared lock as the transaction knows it might update the value after, the transaction request an **update lock**.



8.6.4 Multi-granular locks

First, we need to understand that all locking systems are different. But here we will see one simplified way of a sample locking system.

1. Don't trust transactions to request/release locks
2. Hold all locks until a transaction commits.

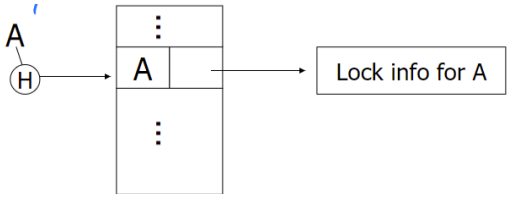


Here, the **scheduler in part 1 locks and unlocks** as it is the only one to have a global view.

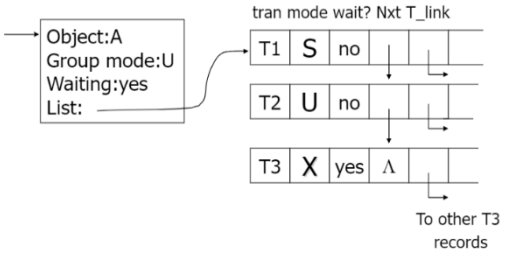
The **scheduler in part 2 order the transaction** to avoid concurrency. And finally, the DB executes the schedule given by part 2.

The **lock table** is really helpful and is composed of a mapping of all possibles object in the DB and if they are locked or not.

For more efficient search, we use **hash tables** that contains all locked data. If an object is not found in the hash table, then it is unlocked.



The **lock info** for an object looks like this (here for A):



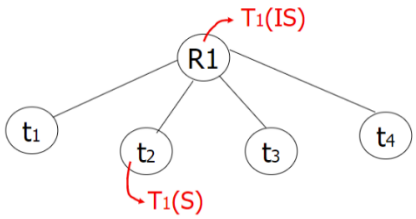
Because we can have multiple locks, there is the Group mode (here U – the most strict lock). And the list is linked to the hash table which is pointing to a linked list.

Note that the transaction order is in FIFO (First In First Out).

Note also that the object can be either a **tuple**, **a relation** or **a disk block**. The bigger the object, the less locks and concurrency there will be.

We talk about **granularity** for levels in the schema. For example in the schema behind, R1 is the higher granularity of t2.

We can see here that as T1 request a shared lock on t2, it has to request an **intentional shared lock T1(IS)** on its **higher granularity level R1**.



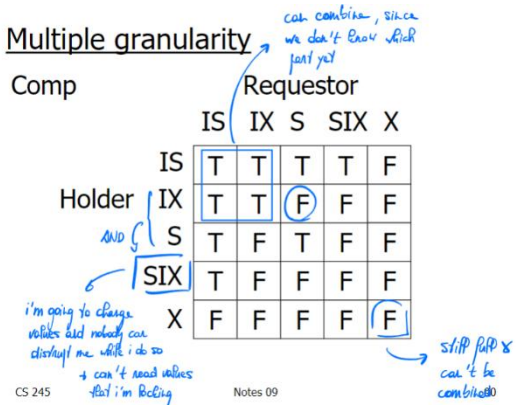
- ➔ If another transaction (say T2) wants to execute a shared lock **T2(S) on R1**, **it can** because **IS** and **S** are **compatible** (see compatibility matrix).
- ➔ But if T2 wants to execute an exclusive lock **T2(X) on t4**, it has to request an **intentional exclusive lock T2(IX)** for R1.

Note that needs to be done recursively if there are more levels.

Multi-granular locks allow **transactions to lock various levels of granularity**. such as entire tables, rows, or even specific fields within a row. This **flexibility** enables a balance between data consistency and system performance.

"But Jamy what are levels of granularity?" – You ask. Well, I have no idea, Fred! But I explained all I could based on my understanding. If it's not enough, well it is not my problem. Maybe you should have listened in class you stupid moron.

Here is the **compatibility matrix**:



Parent locked in	Child can be locked in
IS	S, IS
IX	X, IX, S, IS, SIX
S	/
SIX	X, IX, SIX
X	/

And here are the rules:

1. Follow multiple granularity comp function
2. Lock root of tree first, any mode
3. Node Q can be locked by Ti in S or IS only if parent(Q) locked by Ti in IX or IS
4. Node Q can be locked by Ti in X, SIX, IX only if parent(Q) locked by Ti in IX,SIX
5. Ti is two-phase
6. Ti can unlock node Q only if none of Q's children are locked by Ti

And if you want to do the most unreadable/hard to understand compatibility matrix, you can also add update and increment locks to the party!

Examples

TODO.

8.7 Run a given schedule, and trace the execution steps

This has already been shown in all sections above. Basically, the idea is to follow each action and check the value in the database (for A and B in the examples of this chapter).

The End

Thank you for reading this synthesis (that is almost a syllabus). If you want to thank us, you can send us a nice message on LinkedIn: Raphaël Humblet and Alexandre Achten. We wish you good luck for your exam!