INFOH417 - Database Systems Architecture

Jose Antonio Lorencio Abril

Fall 2022

Professor: Mahmoud Sakr

Student e-mail: jose.lorencio.abril@ulb.be

This is a summary of the course *Database Systems Architecture*, taught at the Université Libre de Bruxelles by Professor Mahmoud Sakr in the academic year 22/23. Most of the content of this document is adapted from the course notes by Sakr, [3], so I won't be citing it all the time. Other references will be provided when used.

Contents

List of Figures

List of Algorithms

Part I
Query Planning: Translating SQL into Relational Algebra

1 Relational Algebra

We are going to start with some definitions:

Definition 1.1. A **relation** is a table whose columns have names, called **attributes**. The set of all attributes is called the **schema of the relation**. The rows of the table are tuples of values for each of the attributes, and are called simply **tuples**. We are going to denote R a relation, and we will express it as $R \sim [A_1, ..., A_n]$ to indicate the schema of the relation, $A_i, i = 1, ..., n$ are the attributes of the schema. If two relations, R and R', share the same schema, we will simply write $R \simeq R'$[a].
A relation is **set-based** if there are no duplicate tuples in it. If this is not the case, the relation is **bag-based**.
A **relational algebra operator** takes as input 1 or more relations and produces as output a new relation. More formally, if we have a set of relations $\Sigma = \{R_1, ..., R_n\} \subset \mathcal{U}$, where $\mathcal{U}$ identifies the set of all possible relations, a relational algebra operator is a function

$$Op : \mathcal{P}(\Sigma) \to \mathcal{U},$$

being $\mathcal{P}(\Sigma)$ the power set of Σ.

[a] Note that the relationship $\simeq$ defines an equivalence relationship whose equivalence groups are all relations with the same schema.

Example 1.1. As an example, we can take $\Sigma = \{StarsIn, MovieStar\}$, $StartsIn = [starName, filmName]$, $MovieStar = [name, birthDate]$. In this case, an operator Op could be such that produces the relation that contains all names of films in which some movie stars in $MovieStar$ born in 1960 participated.

In this example, we have explained what we would like our operator to do, but we need some way to actually compute this. For this, there are some basic operators that can be combined to create complex operators.

1.1 The extended relational algebra

Let's define a set of operators that are useful:

The **union** of two relations with the same schema returns another relation with the same schema and all tuples in any of the two input relations: Let $R_i, R_j \in \Sigma$ such that $R_i \simeq R_j$, then

$$R_i \cup R_j = \{x | x \in R_i \ \vee \ x \in R_j\} \simeq R_i \simeq R_j.$$

Note, nonetheless, that the result of the operator $\cup$ is different in set-based relations than in bag-based relations.

Example 1.2. An example of the operator $\cup$:

$$\begin{bmatrix} A & B \\ 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \bigcup \begin{bmatrix} A & B \\ 3 & 4 \\ 1 & 5 \end{bmatrix} = \left\{ \overset{set-based}{\begin{bmatrix} A & B \\ 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 1 & 5 \end{bmatrix}}, \overset{bag-based}{\begin{bmatrix} A & B \\ 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 1 & 5 \\ 3 & 4 \end{bmatrix}} \right\}$$

The **intersection** of two relations with the same schema returns another relation with the same schema and all tuples in both of the two input relations: Let $R_i, R_j \in \Sigma$ such that $R_i \simeq R_j$, then

$$R_i \cap R_j = \{x | x \in R_i \ \wedge \ x \in R_j\} \simeq R_i \simeq R_j.$$

Example 1.3. An example of the operator $\cap$:

$$\begin{bmatrix} A & B \\ 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \bigcap \begin{bmatrix} A & B \\ 3 & 4 \\ 1 & 5 \end{bmatrix} = \begin{bmatrix} A & B \\ 3 & 4 \end{bmatrix}.$$

The **difference** of two relations with the same schema returns another relation with the same schema and all tuples in the first input relations which don't appear in the second input relation: Let $R_i, R_j \in \Sigma$ such that $R_i \simeq R_j$, then

$$R_i - R_j = \{x | x \in R_i \ \wedge \ x \notin R_j\} \simeq R_i \simeq R_j.$$

Example 1.4. An example of the operator $-$:

$$\begin{bmatrix} A & B \\ 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} - \begin{bmatrix} A & B \\ 3 & 4 \\ 1 & 5 \end{bmatrix} = \begin{bmatrix} A & B \\ 1 & 2 \\ 5 & 6 \end{bmatrix}.$$

The **selection** operator applies a condition on the values of the tuples of the input relation and returns only those tuples that fullfil the condition: Let $R \in \Sigma$ and P a condition, then

$$\sigma_P(R) = \{x | x \in R \ \wedge \ P(R) == true\}.$$

Example 1.5. An example of the operator σ_P:

$$\sigma_{A \geq 3} \left(\begin{bmatrix} A & B \\ 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \right) = \begin{bmatrix} A & B \\ 3 & 4 \\ 5 & 6 \end{bmatrix}.$$

In this case, the condition P is: *'the value of A is bigger than or equal than 3'*.

The **projection** operator returns all tuples of the input relation, but deleting all unspecified attributes: Let $R \in \Sigma$ and $A_{j_1}, ..., A_{j_k} \in [A_1, ..., A_n] \sim R$, then

$$\Pi_{A_{j_1},...,A_{j_k}}(R) = R'_{[A_{j_1},...,A_{j_k}]} = \{y | \exists x \in R \ s.t. \ x(A_{j_1}, ..., A_{j_k}) = y\}.$$

The result of this operation also depends on the type of relations used.

Example 1.6. An example of the operator $\Pi_{[A_{j_1},...,A_{j_k}]}$:

$$\Pi_{A,C} \left(\begin{bmatrix} A & B & C & D \\ 1 & 2 & 3 & 5 \\ 3 & 4 & 3 & 6 \\ 5 & 6 & 5 & 9 \\ 1 & 6 & 3 & 5 \end{bmatrix} \right) = \left\{ \overset{set-based}{\begin{bmatrix} A & C \\ 1 & 3 \\ 3 & 3 \\ 5 & 5 \end{bmatrix}}, \overset{bag-based}{\begin{bmatrix} A & C \\ 1 & 3 \\ 3 & 3 \\ 5 & 5 \\ 1 & 3 \end{bmatrix}} \right\}.$$

The **cartesian product** of two relations with disjoint schemas returns a relation with the schema resulting of combining both schemas and with all possible tuples made out of tuples from the first relation and tuples from the second relation: Let $R_i, R_j \in \Sigma$ such that their schemas are disjoint, then

$$R_i \times R_j = \{z = (x, y) | x \in R_i \ \wedge \ y \in R_j\}.$$

Example 1.7. An example of the operator $\times$:

$$\begin{bmatrix} A & B \\ 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} C & D \\ 2 & 6 \\ 3 & 7 \\ 4 & 9 \end{bmatrix} = \begin{bmatrix} A & B & C & D \\ 1 & 2 & 2 & 6 \\ 1 & 2 & 3 & 7 \\ 1 & 2 & 4 & 9 \\ 3 & 4 & 2 & 6 \\ 3 & 4 & 3 & 7 \\ 3 & 4 & 4 & 9 \end{bmatrix}.$$

The **natural join** of two tuples whose schemas share at most one attribute returns a relation with the schema resulting of combining both schemas and with all possible tuples made out of tuples from the first relation and tuples from the second relation with the condition that they have the same value for the shared attribute: Let $R_i, R_j \in \Sigma$ such that their schemas share at most one attribute, A, then

$$R_i \bowtie R_j = \{z = (x, y) \,|x \in R_i \ \wedge \ y \in R_j \ \wedge x\,(A) = y\,(A)\}\,.$$

Note that if the relations are disjoint, the natural join gives the same results as the cartesian product.

Example 1.8. An example of the operator $\bowtie$:

$$\begin{bmatrix} A & B \\ 1 & 2 \\ 3 & 4 \end{bmatrix} \bowtie \begin{bmatrix} B & D \\ 2 & 6 \\ 3 & 7 \\ 4 & 9 \end{bmatrix} = \begin{bmatrix} A & B & D \\ 1 & 2 & 6 \\ 3 & 4 & 9 \end{bmatrix}.$$

The **theta join** of two relations given a condition P returns all the tuples in the cartesian product of the two relations that fullfil the condition P: Let $R_i, R_j \in \Sigma$ and P a condition, then

$$R_i \bowtie_P R_j = \{x | x \in R_i \times R_j \ \wedge \ P\,(x) == true\} = \sigma_P\,(R_i \times R_j)\,.$$

Example 1.9. An example of the operator $\bowtie_P$:

$$\begin{bmatrix} A & B \\ 1 & 2 \\ 3 & 4 \end{bmatrix} \bowtie_{B=C} \begin{bmatrix} C & D \\ 2 & 6 \\ 3 & 7 \\ 4 & 9 \end{bmatrix} = \begin{bmatrix} A & B & C & D \\ 1 & 2 & 2 & 6 \\ 3 & 4 & 4 & 9 \end{bmatrix}.$$

The **left/right/full outer join** operators are similar to the theta join, but for those tuples in the left-/right/both relation that does not find a match in the other relation, it returns a new tuples with the values of the tuple and the rest of the attributes empty.

Example 1.10. An example of $=\!\bowtie$, $\bowtie\!=$, $=\!\bowtie\!=$:

$$\begin{bmatrix} A & B \\ 1 & 2 \\ 3 & 4 \\ 5 & 5 \end{bmatrix} =\!\bowtie_{B=C} \begin{bmatrix} C & D \\ 2 & 6 \\ 3 & 7 \\ 4 & 9 \end{bmatrix} = \begin{bmatrix} A & B & C & D \\ 1 & 2 & 2 & 6 \\ 3 & 4 & 4 & 9 \\ 5 & 5 & & \end{bmatrix}$$

$$\begin{bmatrix} A & B \\ 1 & 2 \\ 3 & 4 \\ 5 & 5 \end{bmatrix} \bowtie\!=_{B=C} \begin{bmatrix} C & D \\ 2 & 6 \\ 3 & 7 \\ 4 & 9 \end{bmatrix} = \begin{bmatrix} A & B & C & D \\ 1 & 2 & 2 & 6 \\ 3 & 4 & 4 & 9 \\ & & 3 & 7 \end{bmatrix}$$

$$\begin{bmatrix} A & B \\ 1 & 2 \\ 3 & 4 \\ 5 & 5 \end{bmatrix} =\!\bowtie\!=_{B=C} \begin{bmatrix} C & D \\ 2 & 6 \\ 3 & 7 \\ 4 & 9 \end{bmatrix} = \begin{bmatrix} A & B & C & D \\ 1 & 2 & 2 & 6 \\ 3 & 4 & 4 & 9 \\ 5 & 5 & & \\ & & 3 & 7 \end{bmatrix}$$

The **renaming operator** changes the name of a relation, $\rho(R) = R'$. The **feature rename operator** changes the name of an attribute in a relation, $A \to A'$.

The **aggregation operator** of a relation returns another relation in which the tuples that share the value of the aggregating attribute are merged using an **aggregate function**: Let $R \in \Sigma$, with $R \sim [A_1, ..., A_n, A]$, A the aggregating attribute and $f_1, ..., f_n$ the aggregating functions for the rest of the attributes, then

$$\gamma_{A,f_1(A_1),...,f_n(A_n)}(R) = \{z = (v_A, f_1(A_1^{v_A}), ...f_n(A_n^{v_A})) \,|v_A \in R(A)\},$$

where $A_j^{v_A}$ is a short notation for

$$A_j^{v_A} = \Pi_{A_j}(\sigma_{A=v_A}(R)),$$

i.e. all values in $R\left(A_j^{v_A}\right)$ such that come from a tuple whose values for the attribute A is v.

Example 1.11. An example of the operator $\gamma_{A,f_1(A_1),...,f_n(A_n)(R)}$:

$$\gamma_{A,min(B)} \begin{bmatrix} A & B \\ 1 & 2 \\ 1 & 1 \\ 3 & 7 \\ 3 & 9 \\ 4 & 4 \end{bmatrix} = \begin{bmatrix} A & min(B) \\ 1 & 1 \\ 3 & 7 \\ 4 & 4 \end{bmatrix}.$$

1.2 Relational algebra expressions

Now, we can build expressions in relational algebra to get new relations from current ones. Let's return to Example 1.1, we can define the operator Op such that produces the relation that contains all names of films in which some movie stars in $MovieStar$ born in 1960 participated as:

$$R' = Op(MovieStar, StarsIn) = \Pi_{filmName}(\sigma_{birthDate.year=1960}(MovieStar \bowtie_{name=starName} StarsIn)).$$

Relational algebra is the theoretical basis of the **SQL language**, meaning SQL is designed as an implementation of the relational algebra operators that we have seen so far. The equivalent SQL sentence to the last RA operator, OP, is

```
SELECT filmName
FROM StarsIn
JOIN MovieStar ON name=starName
WHERE birthDate.year = 1960;
```

As explained in [4], translating an arbitrary SQL query into a **logical query plan**, or, equivalently, a relational algebra expression, is a complex task. Let's first give some examples.

Example 1.12. We are going to work with some examples now. Let's our database have the following relations:

- Movie(title: string, year: int, length: int, genre: string, studioName: string, producerCERT: int)
- MovieStar(name: string, address: string, gender: char, birthdate:date)
- StarsIn(movieTitle: string, movieYear: string, starName: string)
- MovieExec(name: string, address: string, CERT: int, netWorth: int)
- Studio(name: string, address: string, presCERT: int)

SQL:

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

RA:

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

SQL:

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

RA:

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

At this point, one can understand that it is not easy at all to automatize this procedure of translating from SQL to RA. Not only the process is not trivial as is, but it is also needed to take into consideration that one SQL sentence can be translated into several equivalent RA expressions, which will ultimately be executed in a computer and the election of the translation to execute will affect the efficiency of the program. Let's review the paper [4], explaining each of the translations, assuming set-based relations.

2 Translating SQL into Relational Algebra

2.1 SELECT-FROM-WHERE statemets without subqueries

A query of the form:

```
SELECT select-list
FROM R1 T1,..., Rn Tn
WHERE condition;
```

in which the condition does not involve subqueries, we can translate it as

$$\Pi_{select-list}\left(\sigma_{condition}\left(\rho_{T1}\left(R1\right)\times ...\times\rho_{Tn}\left(Rn\right)\right)\right).$$

2.2 Normalizing WHERE-subqueries into EXISTS and NOT EXISTS form

In general, queries in which there are subqueries in the WHERE clause can arise, and they need to be translated, too. The property used in these cases is that subqueries occurring in the WHERE clause that use the operators $=, <, >, <=, >=, <>, EXISTS, IN, NOT\ EXISTS, NOT\ IN$ or the quantifiers ANY or ALL can all be rewritten to use the operators $EXISTS$ and $NOT\ EXISTS$.

Proposition 2.1. *All conditions using a subquery can be rewritten using only EXISTS and NOT EXISTS.*

Proof. Let's proof some of the results:

- The result using the $EXISTS$ and $NOT\ EXISTS$ operators is obvious.
- Case $= ANY$: a query would look like the following:

```
SELECT select-list
FROM R1
WHERE R1.A = ANY (SELECT B
                  FROM R2
                  WHERE cond);
```

An equivalent query is:

```
SELECT select-list
FROM R1
WHERE EXISTS(SELECT B
                    FROM R2
                    WHERE cond AND R2.B = R1.A);
```

- Case = *ALL*:

```
SELECT select-list
FROM R1
WHERE R1.A = ALL (SELECT B
                          FROM R2
                          WHERE cond);
```

An equivalent query is:

```
SELECT select-list
FROM R1
WHERE NOT EXISTS(SELECT B
                          FROM R2
                          WHERE cond AND R2.B <> R1.A);
```

The rest of the cases $binaryOP + ANY|ALL$ is similar.

- Case *IN*:

```
SELECT select-list
FROM R1
WHERE R1.A IN (SELECT B
                      FROM R2
                      WHERE cond);
```

An equivalent query is:

```
SELECT select-list
FROM R1
WHERE EXISTS(SELECT B
                    FROM R2
                    WHERE cond AND R2.B = R1.A);
```

The case *NOT IN* is analogous. □

Example 2.1. Let's see some examples from the paper:
The query

```
SELECT movieTitle FROM StarsIn
WHERE starName IN (SELECT name
                      FROM MovieStar
                      WHERE birthdate = 1960);
```

is equivalent to:

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

The query

```
SELECT name FROM MovieExec
WHERE netWorth >= ALL (SELECT E.networth
                          FROM MovieExec E);
```

is equivalent to:

```
SELECT name FROM MovieExec
WHERE NOT EXISTS (SELECT E.networth
                  FROM MovieExec E
                  WHERE netWorth < E.netWorth);
```

Without loss of generality, we can now assume that all subqueries in the where clause are of the form *EXISTS* or *NOT EXISTS*.

Now, to translate a query with subqueries, in which an arbitrary number of subqueries inside the subqueries may arise, it seems logical to proceed recursively. The idea is to translate into RA from inner queries to outer queries. For subqueries that do not contain more subqueries, we could translate them as in Section 2.1. The problem in this case is that the subqueries can refer to attributes of relations appearing in the FROM clause of the outer queries. This is known as **correlated queries**.

Example 2.2. A correlated query.

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

The outer relations from which a correlated subquery uses certain attributes are called **context relations**. The attributes of the context relations are the **parameters** of the subquery[1].

2.3 Translating SELECT-FROM-WHERE subqueries

To translate a SELECT-FROM-WHERE statement that is used as a subquery, we must make the following modifications to the method from Section 2.1:

- We must add all context relations to the cartesian product of the relations in the FROM list.
- We must add all parameters as attributes to the projection π.

Example 2.3. The subquery from Example 2.2:

```
SELECT name
FROM MovieStar
WHERE birthdate = 1960 AND name = starName
```

is translated into

$$\Pi_{movieTitle,movieYear,starName,name}\left(\sigma_{birthdate=1960\ \wedge\ name=starName}\left(StarsIn \times MovieStar\right)\right).$$

2.4 De-correlation of subqueries appearing in a conjunctive WHERE condition

Now, let's focus on a particular case:

Suppose we have a query of the general form:

```
SELECT Select-list
FROM from-list
WHERE condition;
```

And the following assumption: *the condition is a conjunction (AND) of SELECT-FROM-WHERE subqueries, possibly with an additional condition that does not contain subqueries*, i.e., the condition is of the form

$$\phi\ AND\ EXISTS(Q_1)\ AND\ EXISTS\,(Q_2)\ AND...\ AND\ NOT\ EXISTS\ (P_1)\ AND...$$

where ϕ denotes the subquery-free condition and $Q_1, ..., Q_n, P_1, ..., P_m$ are select statements. The translation is done in four steps:

[1] Note that not all the parameters must appear in the subquery.

1. Translate ϕ.
2. De-correlate the EXISTS subqueries.
3. De-correlate the NOT EXISTS subqueries.
4. Apply the projection $\Pi_{Select-list}$.

2.4.1 Translating ϕ

It is translated using the method of Section 2.1, but the following context relations must be included:

- All context relations for which parameters occur in ϕ.
- All context relations for which parameters only occur in NOT EXISTS subqueries.

We will obtain a expression of the form

$$\sigma_{\phi}(E),$$

where E is a cartesian product of all the context relations involved. From now on, we are going to adapt and refine E gradually when de-correlating the subqueries.

Example 2.4. Consider the following query, with relations $R(A, B)$ and $S(C)$:

```
SELECT R1.A, R1.B
FROM R R1, S
WHERE EXISTS
        (SELECT R2.A, R2.B
        FROM R R2
        WHERE R2.A = R1.B AND EXISTS
                        (SELECT R3.A, R3.B
                        FROM R R3
                        WHERE R3.A = R2.B AND R3.B = S.C));
```

Let's denote the queries, from outer to inner: Q_1, Q_2 and Q_3. Q_1 does not have a subquery-free part, so we continue with Q_2. The subquery-free part of Q_2 is:

```
SELECT *
FROM R R2
WHERE R2.A = R1.B;
```

So it can be translated as

$$\sigma_{R2.A=R1.B}(\rho_{R2}(R) \times \rho_{R1}(R)).$$

Note that S is a context relation for this subquery-free part, but no parameter from it is needed and it is not only used in NOT EXISTS clauses, so it is not added.

2.4.2 De-correlating EXISTS subqueries

After translating the subquery-free part, we translate all the subqueries $EXISTS(Q_i)$ as explained in Section 2.3, obtaining an algebra expression E_{Q_i}.

Let $A_1, ..., A_p$ be the list of parameters of context relations of Q_i. We can translate $EXISTS(Q_i)$ by joining E with the space of parameters for E_{Q_i}, namely $\Pi_{A_1,...,A_p}(E_{Q_i})$:

$$E := E \bowtie \pi_{A_1,...,A_p}(E_{Q_i}).$$

Example 2.5. Let's continue the translation of Q_2 from Example 2.4. Now, we have to translate Q_3 as:

$$\sigma_{R3.A=R2.B \,\wedge\, R3.B=S.C}(\rho_{R3}(R) \times \rho_{R2}(R) \times S).$$

At this point, we have

$$E = \rho_{R_2}(R) \times \rho_{R_1}(R),$$

$$E_{Q_3} = \sigma_{R3.A=R2.B \,\wedge\, R3.B=S.C}(\rho_{R3}(R) \times \rho_{R2}(R) \times S),$$

and by joining E and E_{Q_3} on the parameters of Q_3 we ensure that we are taking the correct tuples from E and E_{Q_3}. In particular, we are taking the tuples in R_1 for which tuples in R_2, R_3 and S exist that satisfy the requirements of Q_2:

$$\rho_{R_2}(R) \times \rho_{R_1}(R) \bowtie \Pi_{R_2.A,R_2.B,S.C}(\sigma_{R3.A=R2.B \wedge R3.B=S.C}(\rho_{R3}(R) \times \rho_{R2}(R) \times S)).$$

Note that this expression can be simplified:

$$E := \rho_{R_1}(R) \bowtie \Pi_{R_2.A,R_2.B,S.C}(\sigma_{R3.A=R2.B \wedge R3.B=S.C}(\rho_{R3}(R) \times \rho_{R2}(R) \times S)),$$

because we are joining R_2 with a subset of itself, so we will obtain the entire subset.

Remark 2.1. This simplification can always be done. Before joining with $\Pi_{A_1,...,A_p}(E_{Q_i})$, we can remove from E all context relations for Q_i, because they are already present in the parameter space. This way, denoting by $\hat{E}$ the adapted E, we can change what we explained later for

$$E := \hat{E} \bowtie \Pi_{A_1,...,A_p}(E_{Q_i}).$$

Example 2.6. Now we can translate Q_2 as follows:

$$E_2 := \sigma_{R_2.A=R_1.B}(E) =$$

$$\sigma_{R_2.A=R_1.B}(\rho_{R_1}(R) \bowtie \Pi_{R_2.A,R_2.B,S.C}(\sigma_{R3.A=R2.B \wedge R3.B=S.C}(\rho_{R3}(R) \times \rho_{R2}(R) \times S))).$$

Notice how R_2 has been removed from the cartesian product of the subquery-free part of Q_2 that we translated in the first of the examples.

Finally, the translation of the entire Q_1 is

$$\Pi_{R_1.A,R_1.B}(E_2),$$

where $\rho_{R_1}(R)$ and S have been removed from the cartesian product originating from the translation of the subquery-free part of Q_1 (the FROM clause).

2.4.3 De-correlating NOT EXISTS subqueries

Now we can de-correlate the $NOT\ EXISTS(P_j)$ subqueries. We start translating P_j into a RA expression E_{P_j}. Again, we consider the parameters $A_1, ..., A_p$ of the context relations of P_j. The difference now is that we don't join E and E_{P_j}, but we perform an anti-join:

$$E := E \overline{\bowtie} \Pi_{A_1,...,A_p}(E_{P_j}),$$

where

$$R \overline{\bowtie} S = R - (R \bowtie S).$$

In this anti-join, it necessary that R contains all attributes of S, and this is the reason why it is needed to add all context relations appearing only in NOT EXISTS clauses to the cartesian product of the subquery-free part of the query.

2.4.4 Translating the Select-list

Finally, we apply the projection $\Pi_{Select-list}$.

2.5 Flattening subqueries in bag-based relations

Until now, we have supposed that all relations involved are set-based, but this is not the case in real databases, where duplicates can occur. In this case, the requirements for flattening into a normal join are:

- 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.

- The subquery is not under an OR.
- The subquery is not in the SELECT list of the outer query block.
- The subquery type is EXISTS, IN, or ANY, or it is an expression subquery on the right side of a comparison operator.
- 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.
- 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.

Part II
Query Optimization

3 System-R

In this section, we are going to explain System R, which is a pioneering SQL system developed by IBM Research and which was released in 1976, with an accompanying paper, [1].

System R was an experimental prototype database management system, with complete capability, including application programming, query capability, conccurent access support, system recovery, etc.

3.1 Architecture Components

System R is composed by several parts:

- **Relational Storage Interface (RSI)**: internal interface which handles access to single tuples of base relations.
- **Relational Storage System (RSS)**: the supporting system of the RSI. It is a complete storage subsystem in the sense that it manages devices, space allocation, deadlock detection,... It maintain indexes on selected fields of base relations, and pointer chains across relations.
- **Relational Data Interface (RDI)**: the external interface that can be called directly from a programming language. The **SQL language** is embedded within the RDI, and is used as the basis for all data definition and manipulation.
- **Relational Data System (RDS)**: supports the RDI, provides authorization, integrity enforcement and support for alternative views of data. 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 RSS. The RDS also maintains a set of **catalog relations** which describe the other relations, views, images, links, assertions, and triggers known to the system.

A logical diagram of this architecture is depicted in Figure 1.

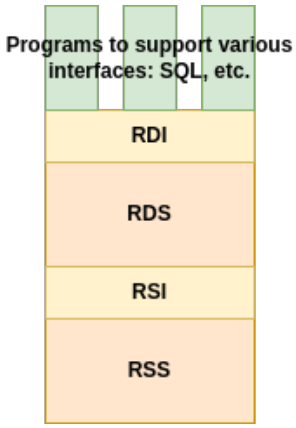


Figure 1: Architecture of System R

3.2 Query language

3.2.1 Data manipulation

The RDI interfaces SQL to a host programming language by means of a concept called a **cursor**, which is a name used at the RDI to identify a set of tuples called its **active set**, and to maintain a position on one tuple of the set. The cursor is associted with a set of tuples by means of the RDI operator **SEQUEL**; the tuples may

EMP		
NAME	SAL	JOB
"John"	1000	"CEO"
"Mike"	800	"PROGRAMMER"
"Sarah"	810	"PROGRAMMER"

Active Set	
NAME	SAL
"Mike"	800
"Sarah"	810

Figure 2: Result of the program. Stored data in EMP (left). Active Set (right).

then be retrieved, one at a time, by the RDI operator **FETCH**. The program must first give the system the addresses of the program variables to be used by means of the RDI operator **BIND**.

Example 3.1. Here, the host program identifies variables X and Y to the system and then issues a query whose results are to be placed in these variables:

```
CALL BIND('X', ADDR(X));
CALL BIND('Y', ADDR(Y));
CALL SEQUEL(C1, 'SELECT NAME:X, SAL:Y
                 FROM EMP
                 WHERE JOB = "PROGRAMMER"');
CALL FETCH(C1);
```

The SEQUEL operator is associating the cursor C1 with the set of tuples which satisfy the query and positioning it just before the first such tuple. The optimizer is invoked to choose an access path whereby the tuples may be materialized, but no tuples are actually materialized in response to the SEQUEL call. The materialization is done as they are called for, one at a time, by the FETCH operator. Each call to FETCH deliver the next tuple of the active set into program variables X and Y. In Figure 2 we can see an example of the stored data in the relation EMP and the resulting Active Set. In this case, after calling FETCH(C1), the values of the variables would be X="Mike" and Y=800. If another call to FECTH(C1) were made, then the variables would be overrided to X="Sarah" and Y=810.

The **DESCRIBE** operator returns the degree and the data types of the active set. The degree is the number of attributes. It is useful when this information is not known in advanced, so it can be inputted to the FETCH operator.

The operator **OPEN** is used to associate a cursor with an entire relation.

Each cursor remains active until an RDI operator **CLOSE** or **KEEP** is issued on it. CLOSE deactivates the cursor, while KEEP causes the tuples identified by a cursor to be copied to form a new permanent relation in the database.

The operator **FETCH_HOLD** is as FETCH, but it also acquires a *hold* on the tuple returned, which prevents other users from updating or deleting it until it is explicitly released by the **RELEASE** operator or until the holding transaction has ended.

3.2.2 Data definition

The SQL statement **CREATE TABLE** is used to create a new base relation. For each field, the field name and data type are specified. When a relation is no longer useful, it may be deleted by issuing a **DROP TABLE** statement.

Access paths include images and binary links. **Images** are value orderings maintained on base relations by the RSS, using multilevel index structures[2], associating a value with one or more **tuple identifiers (TIDs)**, which are internal addresses allowing rapid access to a tuple. One image per relation can have the **clustering property**, which causes tuples whose sort field values are close to be physically stored near each other. **Binary paths** are access paths in the RSS which link tuples in one relation to related tuples of another relation through pointer chains. They are employed in a value dependent manner: the user specifies that each tuple of Relation 1 is to be linked to the tuples in Relation 2 which have matching values in some field/s, and that the tuples on the link are to be ordered in some way[3]. A link may be declared to have the clustering property.

[2]Images are today called just indexes.
[3]So, binary paths are essentialy join indexes.

A **view** is a relation derived from one or more relations, and can be used in the same way as a base table. It can be defined using the **DEFINE VIEW** statement. Views are updated automatically when changes are made to the base tables on which they are defined. When the statement **DROP VIEW** is issued, the indicated view and all other views defined in terms of it disappear from the system. Modifications to views are only allowed if the tuples of the view are associated one-to-one with tuples of an underlying base relation.

The statement **KEEP TABLE** causes a temporary table to become permanent.

The statement **EXPAND TABLE** is used to add a new field to an existing table.

3.2.3 Data Control

A **transaction** is a series of RDI calls which the user wishes to be processed as an atomic act. A transaction starts when the user issues a **BEGIN_TRANS** statement and ends when **END_TRANS** is called. Save points may be specified by means of the operator **SAVE**. When a transaction is active, the user may go back to the beginning of it, or to any save point using **RESTORE**.

Regarding **authorization**, System R does not require a particular individual to be the DB administrator, but allows each user to create his own data objects by executing the create staatements. The creator of an object has full authorization on it. The user can gran selected capabilities for his objects to other users with the statement **GRANT**.

About **integrity assertions**, any SQL predicate may be stated as an assertion about the integrity of data in a base table or view. When an assertion is made by an **ASSERT** statement, its truth is checked. If true, the assertion is atuomatically enforced until it is explicitly dropped by a **DROP ASSERTION** statement. Assertions may describe the permissible states of the database or the permissible transitions in the database. For this latter purpose, the keywords **OLD** and **NEW** are used in SQL to denote data values before and after modification.

If an assertion is **IMMEDIATE**, it cannot be suspended within a transaction, but is enforced after each data modification. Also, *integrity points* may be established by the SQL **ENFORCE INTEGRITY**.

Triggers are a generalization of the concept of assertion, causing a prespecified sequence of SQL statements to be executed when some triggering event occurs.

3.3 Catalogues

Catalogues are maintained by the RDS, and they describe the information of the relations, views, images, links, assertions and triggers known to the system. Each user may access a set of views of the system catalogs which contain information pertinent to him. Users cannot modify a catalog directly, but it is modified indirectly, when tables are created, an image is dropped, etc. A user can enter commments into his various catalog entries by means of the **COMMENT** statement.

3.4 Cursors

As we have seen, **cursors** are pointers to specific tuples on a resulting table from a query. They can be used to retrieve the values of the tuples individually or to store the tables into the database as permanent relations. Cursors are still used, although they are often a low level feature that is not directly used by users, by it is used by the DBMS to provide higher level features to the user.

In addition, SQL can be used to maniputale either one tuple at a time or a set of tuples with a single command. The current tuple of a particular cursor may be selected for some operation using the predicate **CURRENT TUPLE OF CURSOR**.

Example 3.2. Give a 10% raise to all employees in Dept. 50.

```
CALL SEQUEL('UPDATE EMP
             SET SAL = SAL*1.1
             WHERE DNO = 50');
```

Example 3.3. Individual update.

```
CALL BIND('NEWSAL', ADDR(NEWSAL));
CALL SEQUEL('UPDATE EMP
             SET SAL=NEWSAL
             WHERE CURRENT TUPLE OF CURSOR C');
```

3.5 Clustering images

Clustering images, as we have explained, are images (indexes) that can be used to physically store the data in the same order as it is indexed. At most one image per relation can have the clustering property. The reason is simple: it is not possible to store the same data physically in two different orders.

3.6 Optimizer

The objective of the optimizer is to find a low cost means of executing a SQL statement, given the data structures and access paths available. For this, it attempts to minimize the expected number of pages to be fetches from disk into the RSS buffers. The cost of CPU instructions is also taken into account by means of an adjustable coefficient, H, which is multiplied by the number of tuple comparison operations to convert equivalent page accesses. H is useful to adjust the metric for compute-bounded systems or disk access-bounded systems.

The optimizers follows some steps when it receives a SQL statement:

1. Classify the SQL statement into one of several statement types.
2. Examine the system catalogs to find the set of images and links which are pertinent to the given statement.
3. A rough decision procedure is executed to find the set of reasonable methods of executing the statement.
4. If there is more than one reasonable method, the **expected cost formula** is evaluated for each method, and the minimizing method is choosing.

The following parameters, available in the system catalogues, are taken into account:

R **relation cardinality**: number of tuples

D **number of pages occupied by the relation**

T **average number of tuples per page:**

$$T = \frac{R}{D}.$$

I **image cardinality**: number of distinct sort fields values in a given image.

H **coefficient of CPU cost**: $\frac{1}{H}$ is the number of tuple comparisons which are considered equivalent in cost to one disk page access.

An image **match** a predicate if the sort field of the image is the field which is tested by the predicate.

3.6.1 Simple query optimization

In the case of a simple query on a single relation, the optimizer compares the available images with the predicates of the query, in order to determine which of the following eight methods are available:

Method 1 : use a clustering image which matches a predicate whose comparison operator is '='. The expected cost, C is

$$C = \frac{R}{T \times I},$$

that is: from I values, we want one, so we need to retrieve $\frac{R}{I}$ tuples on average. These fit in $\frac{R}{T \times I}$ pages.

Method 2 : use a clustering image which matches a predicate whose comparison operator is not '='. Assuming half the tuples satisfy the predicate, we have

$$C = \frac{R}{T \times 2}.$$

The idea is the same as before, but now we are assuming to retrieve $\frac{R}{2}$ tuples on average.

Method 3 : use a non-clustering image which matches a predicate whose comparison operator is '='. In this case, we have

$$C = \frac{R}{I},$$

because now we might find only one correct tuple per page.

Method 4 : use a non-clustering image which matches a predicate whose comparison operator is not '='. It is

$$C = \frac{R}{2}.$$

Method 5 : use a clustering image which does not match any predicate. We would scan the image and test each tuple against all predicates. The expected cost is

$$C = \frac{R}{T} + H \times R \times N,$$

where N is the number of predicates. So, we recover R tuples, distributed in $\frac{R}{T}$ pages. In addition to this, we need to perform $R \times N$ comparisons (N predicates per tuple), which are weighted by the coefficient of CPU, H.

Method 6 : use a non-clustering image which does not match any predicate:

$$C = R + H \times R \times N.$$

Method 7 : use a relation scan, where this relation is the only one in its segment and test each tupple agains all predicates:

$$C = \frac{R}{T} + H \times R \times N.$$

Method 8 : use a relation scan, where there are other relations sharing the segment. The cost is unknown, but is greater than $\frac{R}{T} + H \times R \times N$.

The optimizer then chooses a method from this set, according to the following rules:

1. If **Method 1** is available, it is chose.
2. If exactly one among **Methods 2,3,5 and 7** are available, it is chosen. If more than one method is available in this class, the expected cost formulas for these methods are evaluated and the method of minimum cost is chosen.
3. If none of the above methods are available, the optimizer chooses **Method 4**, if available.
4. Else, **Method 6**, if available.
5. Else, **Method 8**.

3.6.2 Join query optimization

In the release paper, only 4 methods are explained, although they say the system takes more methods into account.

Method 1 : use images on join fields. A simultaneous scan of the image on $R1.A$ and the image of $R2.A$. The idea is having two pointers, and advance them coordinately, using the fact that images are ordered to find matches.

Method 2 : sort both relations. $R1$ and $R2$ are ordered using their cluster images and two files, $F1$ and $F2$ are created. $F1$ and $F2$ are sorted on field A. The resulting sorted files are scanned simultaneously and the join is performed.

Method 3 : multiple passes. $R1$ is scanned, storing the pertinent fields into a main memory data structure, W. If space in main memory is available to insert a subtuple, S, it is inserted. If there is no space and $S.A$ is less than the current highest value of A in W, S is discarded. After completing the scan of $R1$, $R2$ is scanned using its clustering image and a tuple S' of $R2$ is obtained. Then, W is checked for the presence of $S'.A$. If present, S' is joined to the appropriate subtuple in W. This process continues until all tuples of $R2$ have been examined. If any $R1$ subtuples were discarded, another scan of $R1$ is made to form a new W consisting of subtuples with A value greater than the current highest. $R2$ is scanned again and the process is repeated.

Method 4 : the TID algorithm. Basically, it works as follows:

(a) Obtain the TIDs of tuples from $R1$ which satisfy additional restrictions to the join. Sort them and store the TIDs in a file $F1$. Do the same with $R2$, storing the TIDs in $F2$.

(b) Perform a simultaneous scan over the images on $R1.A$ and $R2.A$, finding the TID pairs of tuples whose values for A match.

(c) Check each pair $(TID1, TID2)$ to see if $TID1$ is present in $W1$ and $TID2$ is present in $W2$. If they are, the tuples are fetched and joined.

A method cannot be applied unless the appropriate access paths are available. The **performance of a method** depends strongly on the clustering of the relations with respect to the access paths. In the paper, four situation are presented in which the optimizer would decide between the four methods, but they claim to detail the cost formulas on a later paper:

Situation 1 : there are clustering images on both $R1.A$ and $R2.A$, but not no images on $R1.B$ or $R2.C$, which are additional conditions. **Method 1** is always chosen.

Situation 2 : there are non-clustering images on $R1.A$ and $R2.A$, but no images on $R1.B$ or $R2.C$. **Method 3** is chosen if W fits into the main memory buffer at once. Otherwise, **Method 2** is chosen.

Situation 3 : there are clustering images on $R1.A$ and $R2.A$ and non-clustering images on $R1.B$ or $R2.C$. **Method 4** is always chosen.

Situation 4 : there are non-clustering images on $R1.A, R2.A, R1.B$ and $R2.C$. **Method 3** is chosen if W fits into the main memory buffer. Otherwise, **Method 2** is chosen if more than one tuple per disk page is expected to satisfy the restriction predicates. In other cases, **Method 4** is chosen.

3.6.3 Optimized Packages

After analyzing a SQL statement, the optimizer produces an **Optimized Package (OP)** containing the parse tree and a plan for executing the statement.

- If the statement is a query, the OP is used to materialize tuples as they are called for by the FECTH command.
- If the statement is a view definition, the OP is stored in the form of a **Pre-Optimized Package (POP)**, which can be fetched and utilized whenever an access is made via the specified view. If any change is made to the structure of a base relation or to the access paths maintained on it, the POPs of all views defined on that relation are invalidated, and each view must be reoptimized to form a new POP.
- When a view is accessed via the RDI operators OPEN and FETCH, the POP for the view can be used directly to materialize the tuples of the view.

3.7 PostgreSQL in relation to System R

Here, we are going to examine how PostgreSQL is similar or different to the characteristics of System R:

- **Catalog**: PostgreSQL also maintains a catalog, with similar information about the relations, indexes, views,... of the system.
- **Tuple Identifier (TID)**: PostgreSQL also has the concept of TID, being it a pair (b, e), where b indicates the disk block in which the tuple is stored, and e is the position where the tuple starts.
- **Image / Clustering image**: the concept of image is called just index in PostgreSQL, but they are extended. PostgreSQL defines several types of indexes, and not all of them are ordered. For instance, a Hash Index is an unordered index, but a BTree index is an ordered index, which would be the most similar one to the concept of image in System R. PostgreSQL also allows to cluster, but the clustering is made at one point in time, using a particular index: this means that subsequent insertions will not be done to maintain the clustering property[4].

[4] If one wishes to do this, the clustering would need to be redone.

- **View**: views in PostgreSQL are very similar to views in System R. They can be thought as a stored query, very similar to what happens in System R. Views are automatically updated when the underlying tables are updated. The difference comes when one tries to directly modify a view. PostgreSQL has the concept of **Updatable view**, which are views that can be modified with INSERT, UPDATE or DELETE. This views are views that meet the conditions:
  - The defining query of the view must have exactly one entry in the FROM clause, which can be a table or another updatable view.
  - The defining query must not contain one of the following clauses at the top level: GROUP BY, HAVING, LIMIT, OFFSET, DISTINCT, WITH, UNION, INTERSECT, and EXCEPT.
  - The selection list must not contain any window function , any set-returning function, or any aggregate function such as SUM, COUNT, AVG, MIN, and MAX.

  If one tries to modify a updatable view, the system will automatically generate the query that performs the appropriate modification in the base table. Note that there are some columns that are not modifiable: if one tried to modify one of these columns, an error would be raised.
- **Cost-based query optimization**: PostgreSQL also has an optimizer that select the best execution plan among several equivalent plans with the objective of minimizing the expected cost.
- **Access path**: access path specify the path chosen by the system to retrieve the requested tuples from a relation. As we have seen so far, System R has basically five types of access paths: sequential scan, images, binary paths, order-and-scan and TID-scan. PostgreSQL defines a wider variety of access path: sequential scan, order-and-scan, TID-scan, hash indexes, BTree indexes, BitMap indexes,...

4 Query Optimization

As we have seen until now, there are alternative ways to evaluate a given query: there are equivalent RA expression for the same query, and also there are different methods that can physically execute a given query.

An **evaluation plan** defines exactly what algorithm is ued for each operation and how the execution of the operations is coordinated.

4.1 Cost-based query optimization

Cost difference between evaluation plans for a query can be enormous. The general steps in cost-based query optimization are as in System R:

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

The **estimation of the cost** is based on:

- Statistical information about relations.
- Statistics estimation for intermediate result.
- Cost formulae for algorithms, computed using statistics.

4.2 Viewing query evaluation plans

Most database support the **EXPLAIN <QUERY>** statement, which displays the plan chosen by the optimizer, along with the cost estimates that it uses for decision.

Some databases also support **EXPLAIN ANALYSE <QUERY>**, which shows actual runtime statistics found by running the query, in addition to showing the plan.

Some databases show the cost as

$$f..l$$

where f is the cost of delivering the first tuple and l is the cost of delivering all results.

4.3 Generating equivalent expressions

Definition 4.1. Two relational algebra expressions are **equivalent** if the two expressions generate the same set/bag of tuples on every legal database instance.
An **equivalence rule** between two expressions ensure that both expressions are equivalent.

Now, we are going to list some equivalence rules:

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

$$\sigma_{P_1 \wedge P_2}(E) \equiv \sigma_{P_1}(\sigma_{P_2}(E)).$$

2. Selection is commutative:

$$\sigma_{P_1}(\sigma_{P_2}(E)) \equiv \sigma_{P_2}(\sigma_{P_1}(E)).$$

3. In a sequence of projections, where $L_1 \subset L_2 \subset ... \subset L_n$, only the outermost one is needed:

$$\Pi_{L_1}(\pi_{L_2}(...(\pi_{L_n}(E)))) \equiv \Pi_{L_1}(E).$$

4. Selections can be combined with cartesian products and theta joins:

$$\sigma_P(E_1 \times E_2) \equiv E_1 \bowtie_P E_2,$$

$$\sigma_{P_1}(E_1 \bowtie_{P_2} E_2) \equiv E_1 \bowtie_{P_1 \wedge P_2} E_2.$$

5. Theta join operations are commutative

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

6. And they are associative, in a soft manner:
   (a) The natural join is associative:

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

   (b) The thetha join is associative in a soft sense:

$$(E_1 \bowtie_{P_1} E_2) \bowtie_{P_2 \wedge P_3} E_3 \equiv E_1 \bowtie_{P_1 \wedge P_3} (E_2 \bowtie_{P_2} E_3),$$

   where P_3 involves attributes that are present in the three relations.

   When we can decide the order of the joins, we would choose the smaller join to be performed before, so that we compute and store a smaller temporary relation.

7. The selection operation distributes over the theta join operation in the following two situations:
   (a) When all the attributes in P_0 involve only the attributes of one of the expressions being joined:

$$\sigma_{P_0}(E_1 \bowtie_P E_2) \equiv \sigma_{P_0}(E_1) \bowtie_P E_2.$$

   (b) When P_1 involves only the attributes of E_1 and P_2 involves only the attributes of E_2:

$$\sigma_{P_1 \wedge P_2}(E_1 \bowtie_P E_2) \equiv \sigma_{P_1}(E_1) \bowtie_P \sigma_{P_2}(E_2).$$

8. The projection operation distributes over the theta join operation as follows: If P involves only attributes from $L_1 \cup L_2$:

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_P E_2) \equiv \Pi_{L_1}(E_1) \bowtie_P \Pi_{L_2}(E_2).$$

   Similar equivalences hold for outerjoin operations.

9. Union and intersection are commutative:

$$E_1 \cup E_2 \equiv E_2 \cup E_1,$$
$$E_1 \cap E_2 \equiv E_2 \cap E_1.$$

10. 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 $-$:

$$\sigma_P (E_1 \cup E_2) \equiv \sigma_P (E_1) \cup \sigma_P (E_2),$$
$$\sigma_P (E_1 \cap E_2) \equiv \sigma_P (E_1) \cap \sigma_P (E_2),$$
$$\sigma_P (E_1 - E_2) \equiv \sigma_P (E_1) - \sigma_P (E_2),$$
$$\sigma_P (E_1 \cap E_2) \equiv \sigma_P (E_1) \cap E_2,$$
$$\sigma_P (E_1 - E_2) \equiv \sigma_P (E_1) - E_2.$$

12. The projection operation distributes over union:

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

Example 4.1. Pushing selections.

Query: find the names of all instructors in the Music department, along with the title of the courses that they teach.

A first RA expression could be the following:

$$\Pi_{name,title} \left(\sigma_{dpt\_name='Music'} \left(Instructor \bowtie \left(Teaches \bowtie \Pi_{course\_id,title} (Course)\right)\right)\right).$$

It can be transformed using rule 7a:

$$\Pi_{name,title} \left(\left(\sigma_{dpt\_name='Music'} (Instructor)\right) \bowtie (Teaches) \bowtie \Pi_{course\_id,title} (Course)\right).$$

The advantage of doing this is taht by performing the selection as early as possible we are reducing the size of the relation to be joined.

Example 4.2. Pushing projections.

We start with the RA expression:

$$\Pi_{name,title} \left(\left(\sigma_{dpt\_name='Music'} (Instructor) \bowtie Teaches\right) \bowtie \Pi_{course\_id,title} (Course)\right).$$

When we compute

$$\sigma_{dpt\_name='Music'} (Instructor) \bowtie Teaches,$$

we obtain a relation with schema $(ID, name, dpt\_name, salary, course\_id, sec\_id, semester, year)$. Equivalence rule 8 allows to push projections, eeliminating unneeded attributes from intermediate results to get:

$$\Pi_{name,title} \left(\left(\Pi_{name,course\_id} \left(\sigma_{dpt\_name='Music'} (Instructor) \bowtie Teaches\right)\right) \bowtie \Pi_{course\_id,title} (Course)\right).$$

This is useful because performing the projection as early as possible reduces the size of the relation to be joined. Note that *course_id* needs to be projected because it is needed for the join.

Example 4.3. Join ordering.

Consider the expression

$$\Pi_{name,title} \left(\left(\sigma_{dpt\_name='Music'} (Instructor) \bowtie Teaches\right) \bowtie \Pi_{course\_id,title} (Course)\right).$$

In this case, we could compute

$$Teaches \bowtie \Pi_{course\_id,title} (Course)$$

first, and then join the result with the left relation. The problem with this approach is that doing this join first seems more likely to be large, as only a small fraction of the university's instructor are going to be from the Music department. So it is better to leave the query as is.

4.4 Enumeration of equivalent expressions

Query optimizers use equivalence rules to systematically generate expressions equivalent to the given expression, which is a first translation of the query.

All the equivalent expressions can be generated with the following approach:

```
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
```

4.5 Cost estimation

The optimizer takes into account the cost of each operator and the statistics of the input relations, such as the number of tuples and the sizes of the tuples. Also, inputs can be results of sub-expressions, so we need to estimate estatistics of these results. For this purpose, more statistics, such as the number of distinct values for an attribute, are used.

4.6 Choice of execution plan

Once we have generated different equivalent expressions, we need to decide which one to use to execute the query and get the results. For this, we must consider the interaction of evaluation techniques, because choosing the cheapest algorithm for each operation disregarding the others may not yield best overall algorithm. For example, a merge-join may be costlier than a hash-join, but may provide a sorted output which could reduce the cost for an outer level aggregation.

Practical query optimizers incorporate elements of 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.

4.6.1 Best join-order problem

Problem: find the best join-order for

$$R_1 \bowtie R_2 \bowtie ... \bowtie R_n.$$

A first idea could be to check all possibilities and choose the cheapest one. But...

Proposition 4.1. *For the best join-order problem, with n relations involved, there are*

$$\frac{(2\,(n-1))!}{n-1}$$

different possible join orders.

Proof. First, we need to count all possible orderings, i.e. the number of permutations, which is known to be $n!$. Now, we have to count all possible ways to assign the $n-1$ needed parenthesis. This is known to be the Catalan number[5]

$$\#()_{n-1} = C_{n-1} = \frac{(2\,(n-1))!}{n!\,(n-1)!}.$$

Thus, the total amount is

$$n! \cdot C_{n-1} = n! \times \frac{(2\,(n-1))!}{n!\,(n-1)!} = \frac{(2\,(n-1))!}{(n-1)!}.$$

□

This number is huge, and it is unfeasable to check the whole search space. Thus, a different approach is needed.

[5] See Catalan Numbers.

Algorithm 1 procedure findbestplan(S)

```
if (bestplan[S].cost != infty)
        return bestplan[S]
// else it has not been computed yet
if (S contains only 1 relation)
        set bestplan[S].plan and bestplan[S].cost
                based on the best way to access S
else for each non-empty proper subset S1 of S
        P1 = findbestplan(S1)
        P2 = findbestplan(S - S1)
        A = best algorithm for joining P1 and P2
        cost = P1.cost + P2.cost + A.cost

        if cost < bestplan[S].cost
                bestplan[S].cost = cost
                bestplan[S].plan = plan
        return bestplan[S]
```

Dynamic programming approach

Using dynamic programming, the least-cost join for any subset of $\{R_1, ..., R_n\}$ is computed only once and stored for future use. The algorithm works as follows:

- Consider all possible plans of the form
$$S_1 \bowtie (S - S_1),$$
where S_1 is any non-empty subset of S.
- Recursively compute costs for joinin subsets of S to find the cost of each plan. Choose the cheapest of the $2^n - 2$ alternatives.
- **Base case**: single relation access plan.
  - Apply all selections on R_i using best choice of indices on R_i.
- When the plan for any subset is computed, we store it and reuse it when it is required again, instead of recomputing it.

The pseudocode is shown in Algorithm 1.

The time complexity of this algorithm is $O(3^n)$ and the space complexity is $O(2^n)$. This is a huge gain with respect to checking the whole search space, but it is still a very high cost.

Left-deep join trees

In left-deep join trees, the right-hand-side input for each join is a relation, not the result of an intermediate join. In Figure 3 we can see an example of what is a left-deep join tree, and what is not.

With this structure, we can reduce the cost of the optimization problem.

For a set of n relations, we can 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 the best join order is in this case $O(n2^n)$ and the space complexity remains the same.

At this points, one might think that it is of no use bothering with optimizing the order of queries, if it so costful, but, in reality, typical queries have a small n, usually less than 10, and a good ordering can change a query from being unfeasable to being executed in an acceptable time.

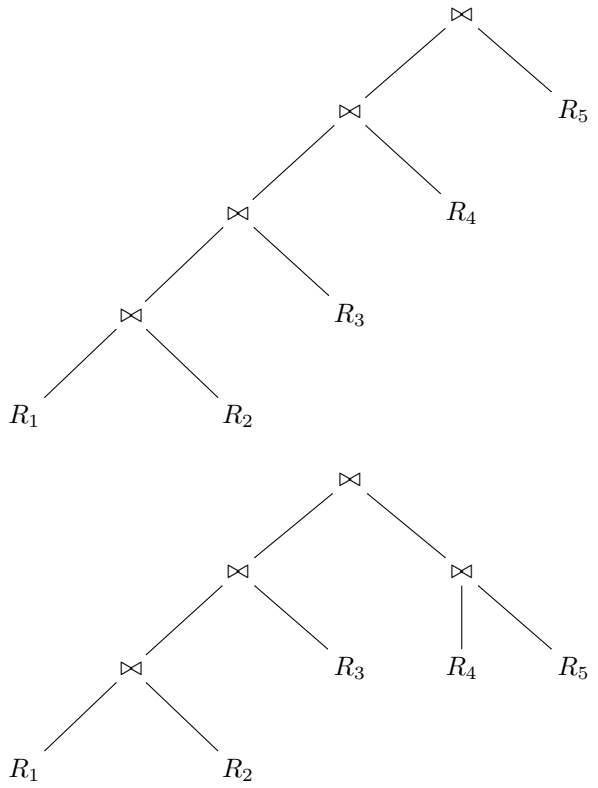


Figure 3: A left-deep join tree (top) and not a left-deep join tree (bottom).

Heuristic optimization

As we have seen, cost-based optimization is expensive, even using dynamic programming.

Heuristic optimization transforms the query-tree by using a set of rules that typically improve execution performance. These rules can include:

- Perform selection early.
- Perform projection early.
- Peform most restrictive selection and join operations before other similar operations.

Some systems use only heuristics, while others combine the two approaches. A frequently used approach is the following:

1. Heuristic rewriting of nested block structure and aggregation.
2. A cost-based join-order optimization for each block.

There is usually an **optimization cost budget** to stop optimization early if the cost of the plan is less than the cost of the optimizations to be made.

Also, it can be useful to implement **plan caching** to reuse previouly computed plans if queries are resubmitted.

It is worth to note that even with the use of heuristics, cost-based query optimization imposes a substantial overhead in the computations, but it is worthy for expensive queries. For this reason, optimizers often use simple heristics for cheap queries, and perform a more exhaustive enumeration for more expensive queries.

5 Statistics for cost estimation

Statistics of relations are of great importance for improving the performance of the system, because they allow to estimate the cost more accurately.

Some statistical information that is used:

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

5.1 Histograms

Histograms are useful for cost estimation. Histogram can be of two types:

- **Equi-width**: the space is divided into M buckets of the same size.
- **Equi-depth**: the space is divided into M buckets, in such a way that all buckets have the same number of tuples inside.

Many databases store the n most frequent values and their counts, and they construct histogram for the remaining values. Usually, they are computed not on all the actual values, but on a sample of them.

This sampling approach make it possible for the statistics to be outdated, so they need to be recomputed:

- Some databases require a ANALYZE (VACUUM) command to be explicitly executed to update statistics.
- Others automatically perform the recomputation.

5.2 Estimation of selection size

We want to estimate the size of a selection, but this depends on the conditions to fullfil:

- $\sigma_{A=v}(r)$: a simple equality condition size is estimated as the size of the relation, divided by the number of distinct values for attribute A:
$$C = \frac{n_r}{V(A, r)}.$$
Nonetheless:
  - If the attribute is a key attribute: the estimation is 1.
  - If the value v is among the most frequent values of attribute A, we can give more accurate estimations.
- $\sigma_{A \leq v}(r)$: a simple inequality condition is more complex than before. Let c denote the estimated number of tuples satisfying the condition. If $m = \min(A, r)$ and $M = \max(A, r)$ are available in the catalog then:
$$c = \begin{cases} 0, & if\ v < m\ \vee\ v > M \\ n_r \frac{v-m}{M-m}, & otherwise \end{cases}$$
So, we approximate by a linear interpolation over the total number of records. Note that if histograms are available, this estimation can be refined by summing all buckets below v and interpolating only in the bucket in which v lies.
When there is a lack of statistical information, c is assumed to be $\frac{n_r}{2}$.

For more complex conditions, we need a new definition:

Definition 5.1. The **selectivity** of a condition P is the probability that a tuple in the relation r satisfies P. If s_P is the number of tuples satisfying the condition, then the selectivity is given by

$$S_P = \frac{s_P}{n_r}.$$

- **Conjunction**, $\sigma_{P_1 \wedge ... \wedge P_k}(r)$: assuming independence, we have

$$c = n_r \times S_{P_1} \times ... \times S_{P_k} = n_r \frac{s_{P_1} \cdot ... \cdot s_{P_k}}{n_r^n}.$$

- **Disjunction**, $\sigma_{P_1 \vee ... \vee P_k}(r)$: we do the following

$$Prob(A_1 \vee ... \vee A_k) = Prob\left(\overline{\overline{A_1 \vee ... \vee A_k}}\right) = Prob\left(\overline{\overline{A_1} \wedge ... \wedge \overline{A_k}}\right) = 1 - Prob\left(\overline{A_1} \wedge ... \wedge \overline{A_k}\right),$$

  which, assuming independence is equal to

$$1 - Prob\left(\overline{A_1}\right) \cdot ... \cdot Prob\left(\overline{A_k}\right) = 1 - (1 - Prob(A_1)) \cdot ... \cdot (1 - Prob(A_k)).$$

  So, it is

$$c = n_r \times \left[1 - \left(1 - \frac{s_{P_1}}{n_r}\right) \cdot ... \cdot \left(1 - \frac{s_{P_k}}{n_r}\right)\right].$$

- **Negation**, $\sigma_{\neg P}(R)$: this is just

$$c = n_r \cdot (1 - S_P) = n_r - n_r S_P = n_r - s_p.$$

5.3 Estimation of the size of joins

The **cartesian product** of two relations r_1 and r_2 contains exactly $n_{r_1} \times n_{r_2}$ tuples, and each tuple occupies $l_{r_1} + l_{r_2}$ bytes.

About the join:

- If $r_1 \cap r_2 = \emptyset$, then $r_1 \bowtie r_2$ is the same as $r_1 \times r_2$.
- If $r_1 \cap r_2$ is a key for r_1, then a tuple of r_2 will join with at most one tuple from r_1, so the number of tuples in $r_1 \bowtie r_2$ is, at most, the number of tuples in r_2.
- If $r_1 \cap r_2$ is a foreign key in r_2 referencing r_1, then the number of tuples in $r_1 \bowtie r_2$ is exactly the same as the number of tuples in r_2.
- If $r_1 \cap r_2 = \{A\}$ is not a key for r_1 nor r_2: let assume that every tuple t in r_1 produces tuples in the join, then the number of tuples in the join is estimated to be

$$c = \frac{n_{r_1} \cdot n_{r_2}}{V(A, r_2)},$$

  i.e., the number of tuples in the cartesian product, divided by the number of different values for the attribute A in the second relation. This is because each value in r_1 will join with more or less $\frac{n_{r_2}}{V(A,r_2)}$ values.

  If we assume the reverse, i.e., every tuple in r_2 produces tuples, then we get

$$c = \frac{n_{r_1} \cdot n_{r_2}}{V(A, r_1)},$$

  for the same reason.

  The lower of these two estimates is probably the most accurate one, so both are computed and the best one is chosen.

  This estimates can be improved using histograms, by using the same formula but on each bucket, and summing them up.

Example 5.1. Estimating the size of a join.
Let perform

$$student \bowtie takes$$

with the following information:

- $n_{student} = 5000, f_{student} = 50$ so $b_{student} = \frac{5000}{50} = 100$.
- $n_{takes} = 10000, f_{takes} = 25$ so $b_{takes} = \frac{10000}{25} = 400$.
- $V(ID, takes) = 2500$, so, on average, each student that has taken a course, has taken 4 courses.
- The attribute ID in takes is a foreign key referencing *student*, in which it is a primary key: $V(ID, student) = 5000$.

The most accurate estimation in this case is the one using the fact that ID is a foreign key, which implies that the number of tuples is the same as the number of the referencing relation, *takes*, so

$$c = n_{takes} = 10000.$$

Let's nonetheless compute an estimate disregarding this:

$$c_1 = \frac{5000 \cdot 10000}{2500} = 20000,$$

$$c_2 = \frac{5000 \cdot 10000}{5000} = 10000.$$

We choose the lower estimate, c_2, which in this case is the same as the one we chose before! But this is no surprise, this will always happen with foreign keys, because $V(A, r_1) = n_{r_1}$ if A is key, so we would have

$$c_2 = \frac{n_{r_1} \cdot n_{r_2}}{n_{r_1}} = n_{r_2},$$

which is the same value that we get using the other estimation. Also, $V(A, r_2)$ will be at most n_{r_1}, because it is a foreign key, so it cannot have more values than the referenced attribute!

Part III
Indexing

6 Conventional indexes

Definition 6.1. An **index** is a data structure that facilitates the recovering of data. The idea is to maintain pointers to the specific directions where some data is stored.

As we know, the disk can be logically seen as a sequence of pages of a certain size. Every file that we store in a computer must be stored in one or more pages. Now, imagine we want to retrieve a file's content. For this, we need to fetch the data from where it is stored. If we don't use indexes, we would need to sequentially traverse the disk until we find the desired file.

Example 6.1. Imagine a simplified setup with a disk of N pages and with files that occupy one page. If we need to recover a specific file that is stored in memory, without further information, it would take an average of $\frac{N}{2}$ pages to be fetched.

An index can be used to mitigate this impact. There are multiple types of indexes, but the simplest form of an index is just a map in which each file identifier is associated to the direction of its first byte in memory. This way, only knowing which file we want to recover, we can access it directly using the index.

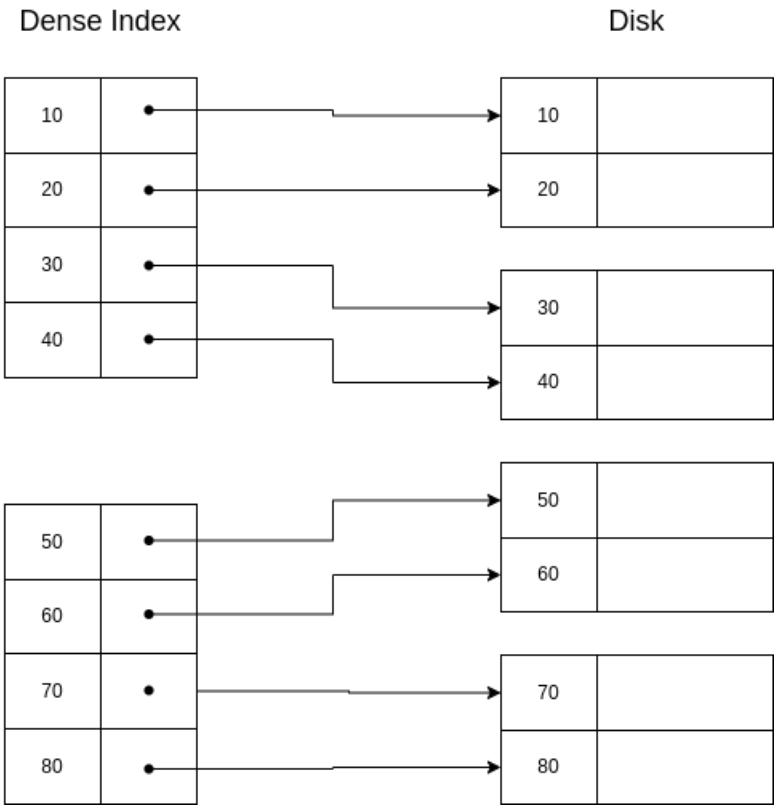
Example 6.2. In the previous setup, imagine we store an index in the first page. In this scenario, to recover a specific file we need to fetch the first page, look the index to get the direction of our file, and directly fetch the correct page. In total, we would fetch 2 pages.

Definition 6.2. A **dense index** is an index that maintains one pointer per key.

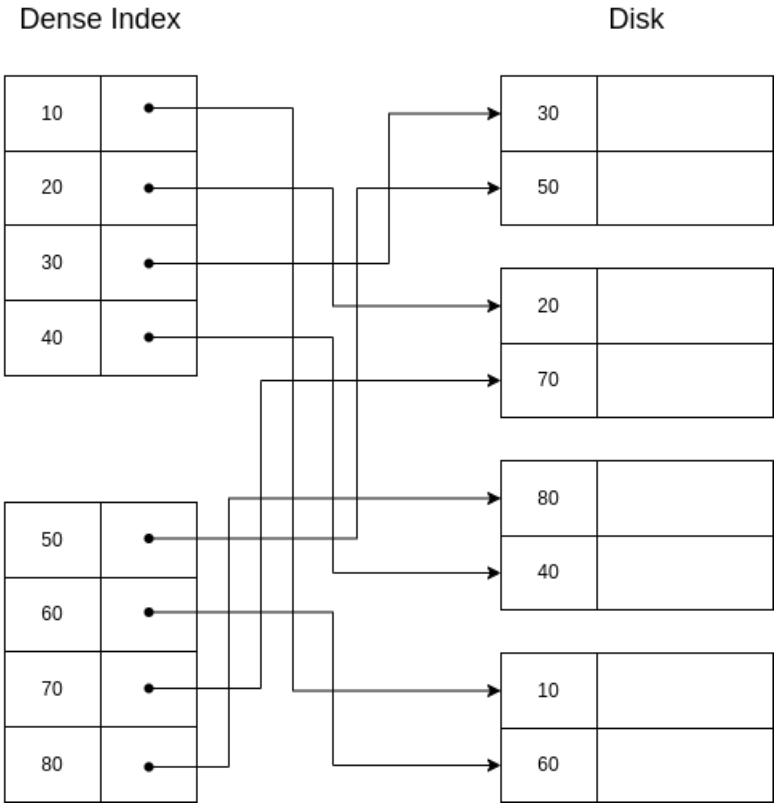
Properties of dense indexes:

- If a key is not listed, the key does not exist.
- They can be used in sequential and non-sequential files.
- Querying a dense index is more efficient than querying a sequential file because we will likely retrieve less pages from disk to memory.
- They are usually ordered, so searchs can be done using the binary search algorithm.

Example 6.3. A dense index.
A dense index on a sequential file looks like this:



And on a non-sequential file like this:



If we want to retrieve the segment with key=30, we would do binary search in the index and then we would get the disk direction of this segment.

If we are asked to retrieve a segment with a key that is not in the index, we would directly return an error after searching for it and not finding it, because we are sure there is no segment with that key in the disk. For

example, there is no segment with key 25, and so there is no entry in the index with key 25.

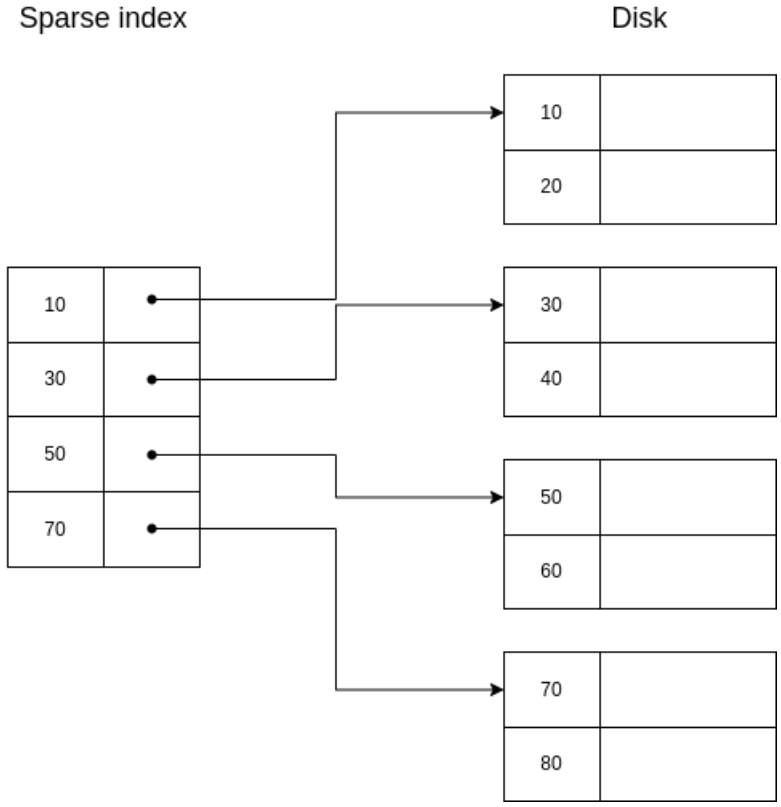
Definition 6.3. A **sparse index** maintains a pointer per page/block. This means that only the first key on the each block.

Properties of sparse index:

- They are also sorted, so binary search can be conducted to find the requested key. In this case, we are looking for the biggest key that is less than the requested key. We then would go to the page where this key is and scan sequentially until we reach the requested key or a key bigger than it, in which case we would return an error.
- They need less space than dense indexes.
- Can only be used in sequential files, because the basic idea for sparse indexes to work is that data is sorted in the same way as the index.

Example 6.4. Sparse index.

A sparse index looks like this:



If we want to retrieve the segment with key=30, the procedure is exactly as with a dense index.

If we want to retrieve the segment with key=40, then we would search for 30 (the biggest key smaller than 40), we would go to where it is. Then, we would advance until we found the segment with key=40.

If we want to retrieve the segment with key=25, we would search for 10, we would go to where it is. Then, we would advance until we reached the end of the page, and we would return an error, because now we are sure there is no segment with key=25.

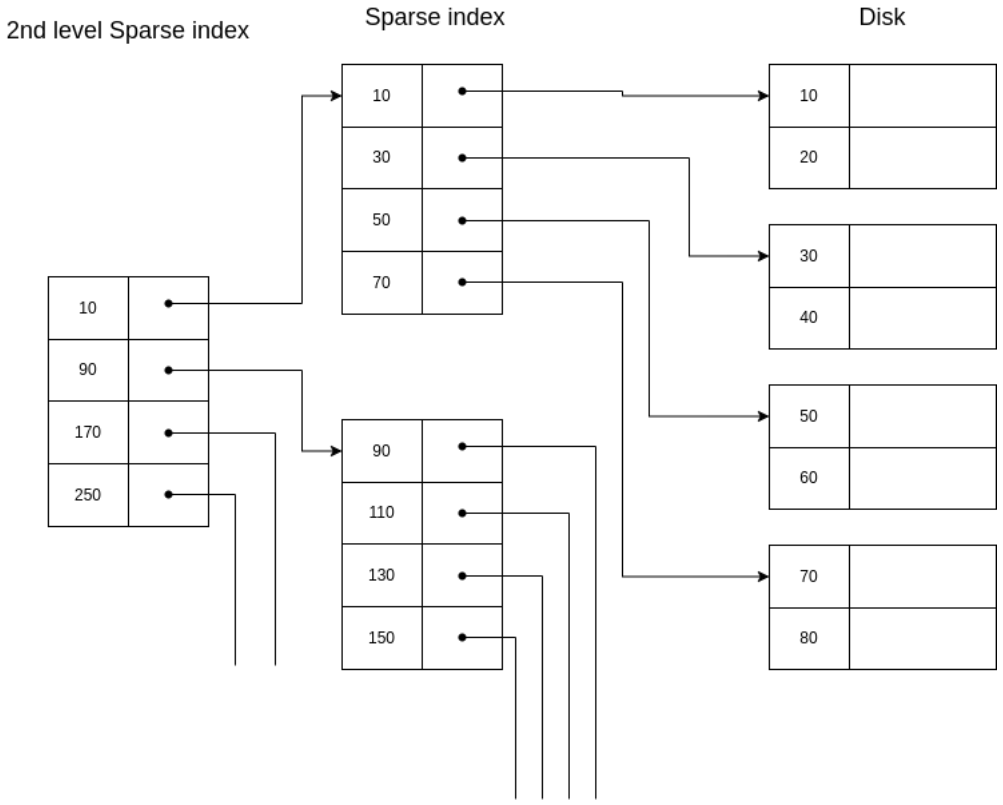
6.1 Sparse second level index

When an index becomes very big, the searchs start to slow down, because the binary search needs to be done over a bigger index, which can even occupy several pages, which would need to be fetched.

When this happen, a possible way to speed up the search is to add a sparse index that points to the already defined index.

Example 6.5. A second level sparse index.

In this diagram we can see a second level sparse index on a sparse index:



If we wanted to retrieve the segment with key=30, we would check the 2nd level sparse index, we would get the page where the index should have this key stored. We would go to this page, and we would find the key=30 already in the index, so we would go to the indicated direction.

Question: does it make sense to use a second level dense index?

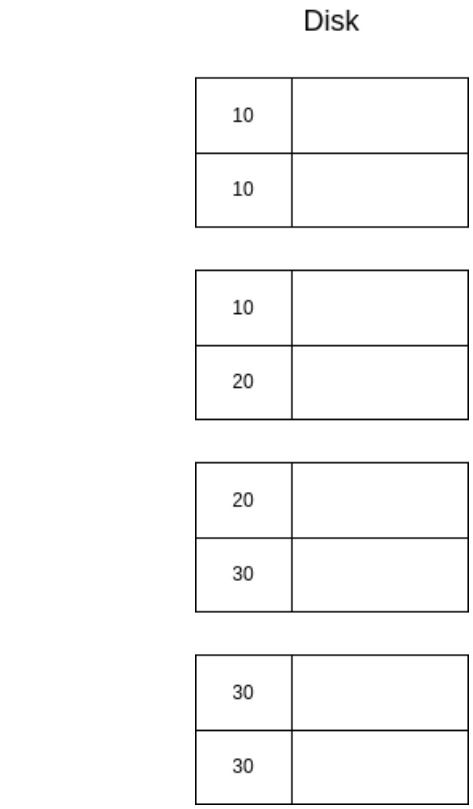
No, it would be a copy of the first level index, so we would not gain anything. Second level indexes are sparse. Note, nonetheless, that the first level index can indeed be of any kind.

Question: what is the tradeoff between sparse and dense indexes?

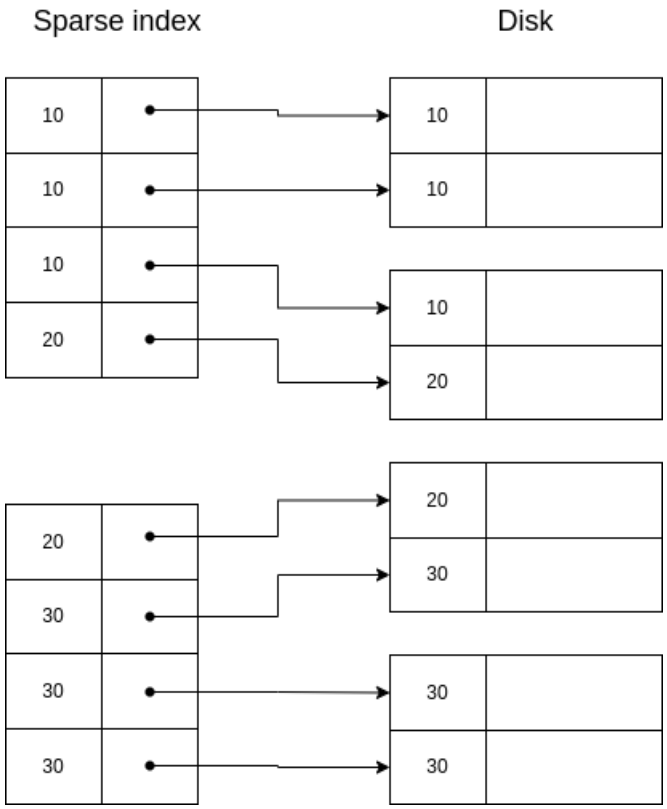
Sparse indexes needs less space to be stored and this also allows to have a bigger part of the index in memory when we need it. On the other hand, dense indexes can tell if any record exists without accessing the files.

6.2 How to deal with duplicate keys.

Imagine we have a disk with the following data:

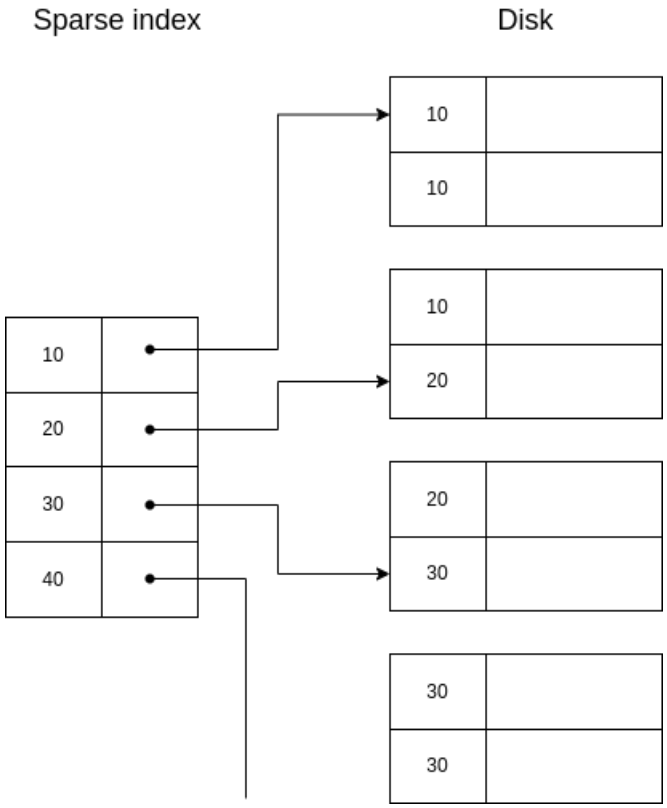


A naïve solution would be to just use a dense index, where all the keys are listed repeatedly:



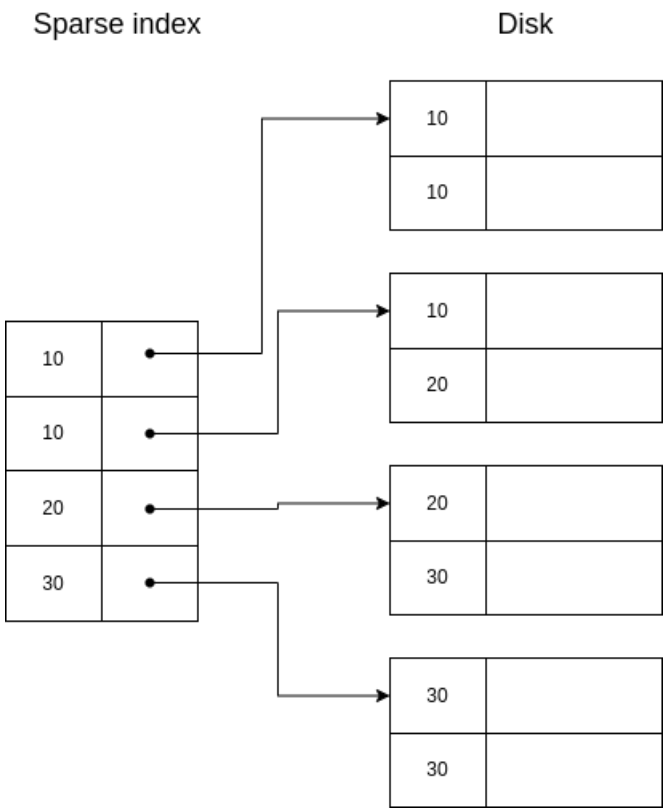
In this case, we are solving the problem of duplicate keys... but we are probably using more space that we wanted. It would be better to have unique keys in the indexes. Thus, a second approach could be to only store

the first appearance of each key, and use it to fetch the data, scanning sequentially until all records with the same key have been retrieved. This is illustrated below:



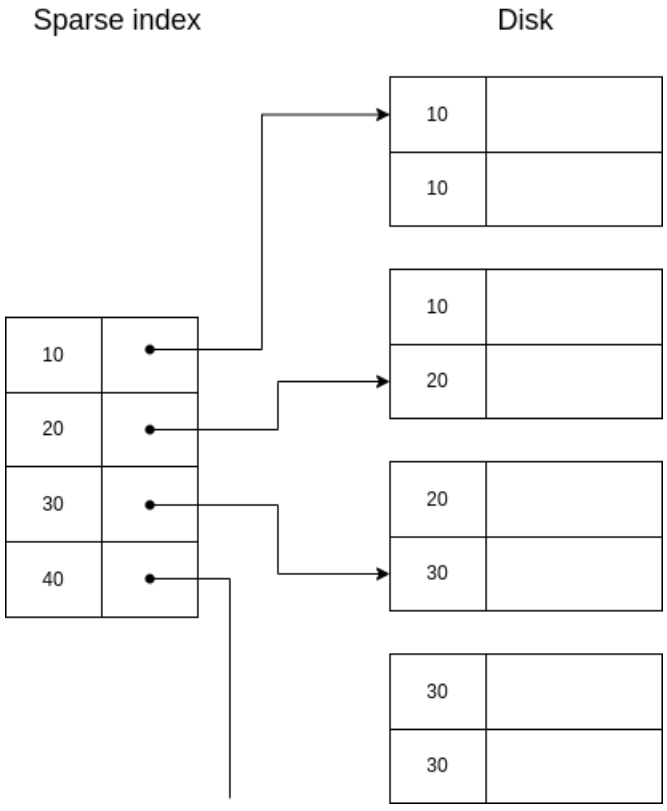
See, nonetheless, that this solution requires that the file is sequentially stored in memory, because when we recover the second segment with key=10, in order to recover the third one, the only possible way is to continue a sequential scan.

Now, a third approach is an intermediate approach: we can use a sparse index with duplicate keys, meaning we index the first key in each page:



In this case, when we want to search for a key, we always need to go to the biggest key that is smaller than the requested one, even if the requested one is in the index keyset. For example, if in the example above we used the indexed direction for key=20, we would miss the first record of this key.

An improved version of this solution is to index only the first new key in each page:



Note that in this case, all keys will be listed once, so we don't need to retrieve the file for inexistent keys, but sequentially ios required.

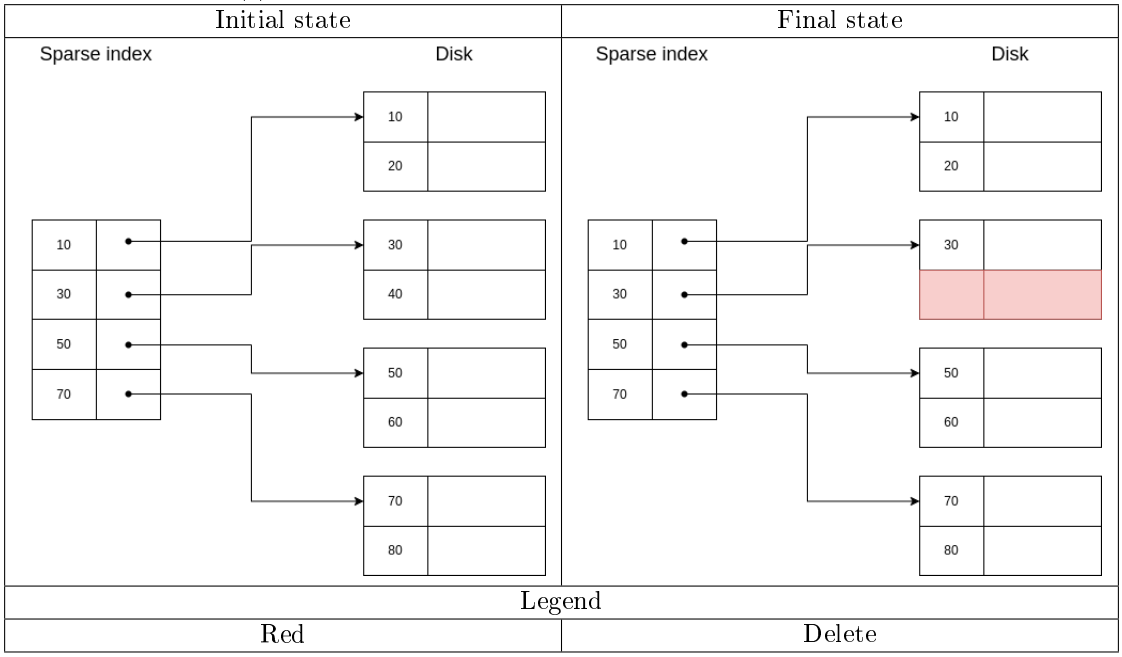
6.3 How to delete records

6.3.1 Deletion from sparse index with no duplicates

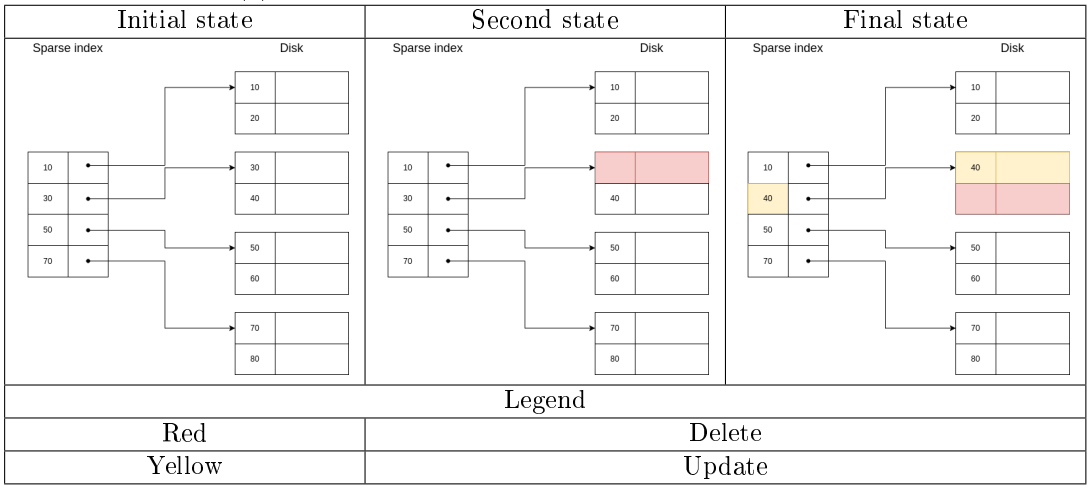
If we want to delete a record, we need to make sure that the index is updated if needed. The steps to delete a record are the following:

1. To delete key K, do a binary search in the index.
2. Depending if K is in the index or not:
   (a) If K is not in the index we visit the direction of the biggest key that is smaller than the requested one. We advance until we find the record with key K and we delete it.
   (b) If K is in the index, we visit its direction and we delete it. Now, two possibilities arise:
      i. If there are more records in the same page, we shift them up, and we update the key.
      ii. If there no more records in the same page, we delete the key from the index and we shift the rest of the keys up.

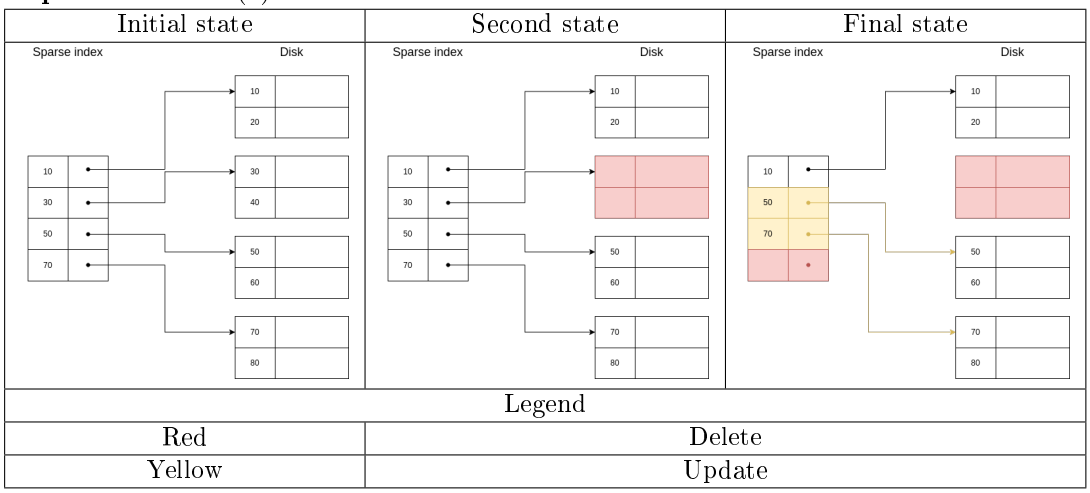
Example 6.6. Case 2.(a): DELETE 40



Example 6.7. Case 2.(b).i.: DELETE 30



Example 6.8. Case 2.(b).ii.: DELETE 30 and 40

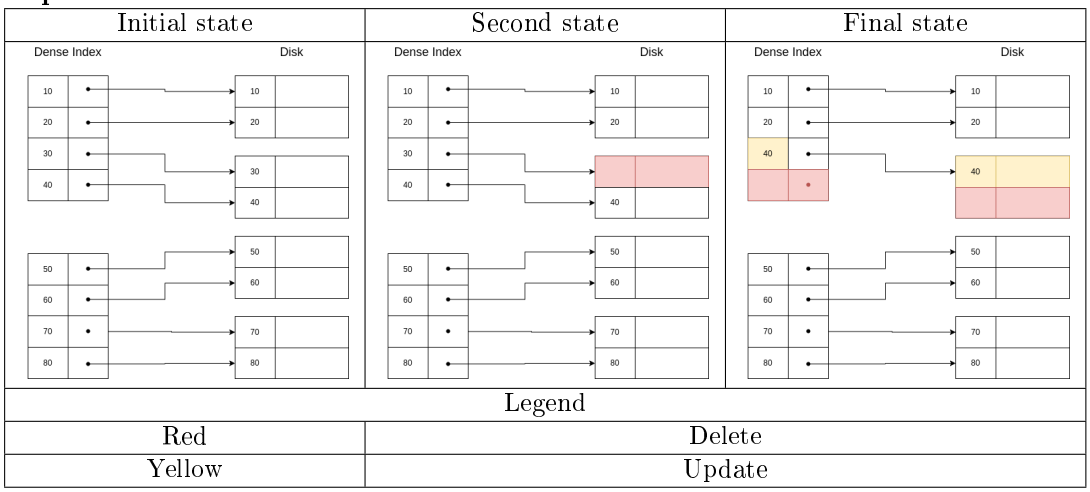


6.3.2 Deletion from dense index

In this case, we will always find the keys to delete in the index, so the steps are easier:

1. To delete key K, do a binary search on the index.
2. Delete records in the corresponding page, shifting up the rest of the records to not leave holes.
3. Update all shifted records.

Example 6.9. Deletion from dense index: DELETE 30



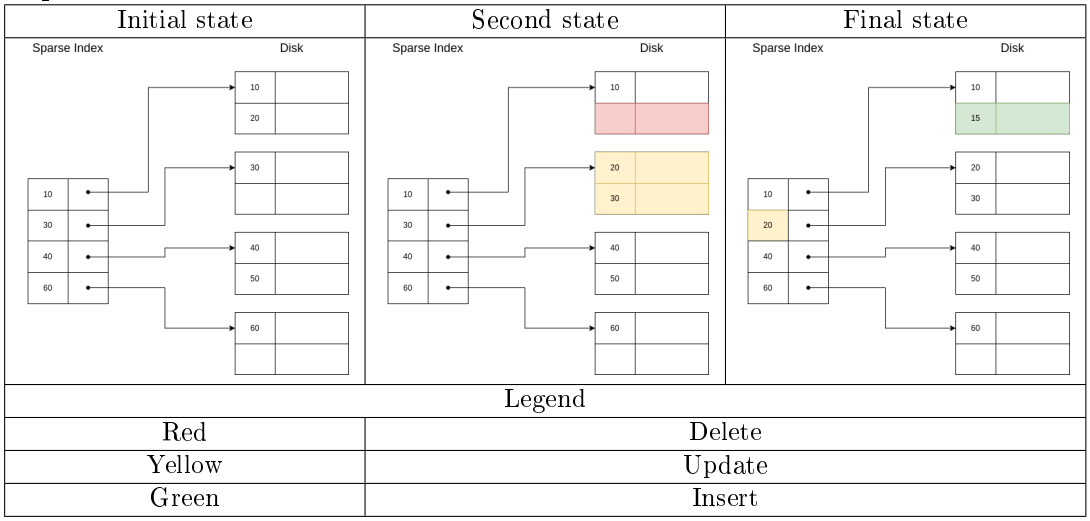
6.4 How to insert records

We need to follow the next steps[6]:

1. We want to insert record with key K. First, we do a binary search to see where it should be located.
2. Now, in the first page that it can be located according to the index, two things can happen:
   (a) If there is space for the record: we insert it.
   (b) If there is not space for the record: we need to shift the following records down, updating the necessary index entries.

[6] The steps are analogous for sparse and dense indexes.

Example 6.10. INSERT 15



Legend	
Red	Delete
Yellow	Update
Green	Insert

6.5 Secondary indexes

Imagine we have an unordered file in memory, which we would like to be able to traverse in order without implying great costs. If we try to do this by sequentially scanning the disk, we would need to fetch several times each page and it would be highly inneficient, so we could think on using indexes to solve this problem.

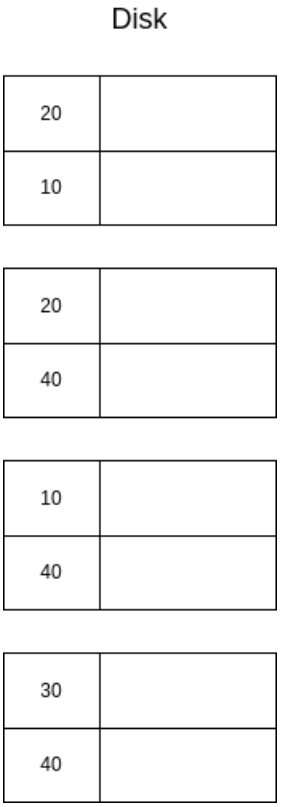
As we have seen, sparse indexes cannot be used with unordered files (some records would be lost), so our only option here is to use an ordered dense index that enables us to recover each record in the desired order. Now, as we are indexing the whole file with an dense index, it is likely that the index is huge, so it seems convenient to add a second level sparse index to speed things up even more.

This is a secondary index:

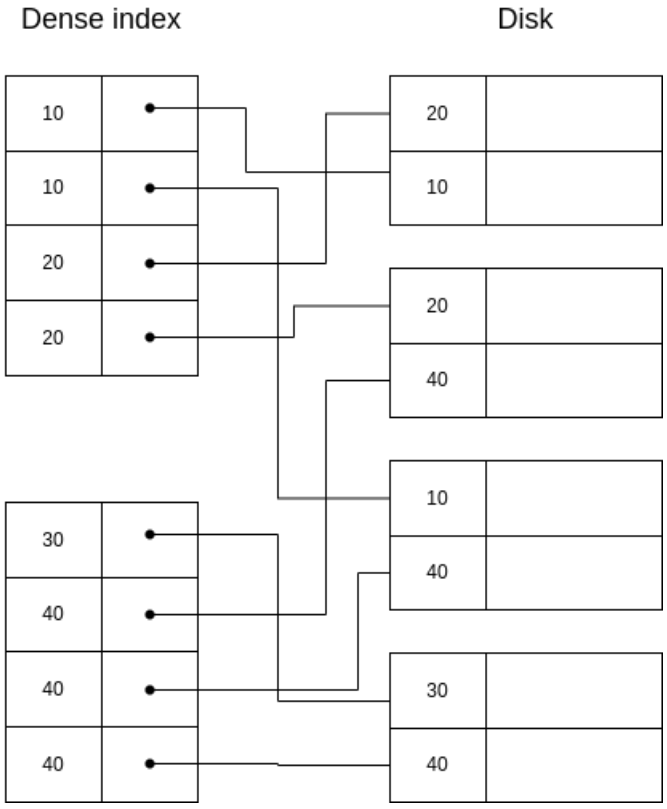
Definition 6.4. A **secondary index** is a N-level index structure, composed by a first-level dense index and the rest of the levels are sparse. All this indexes are ordered to make use of binary search, in order to be capable of recovering unsequentially stored records from disk efficiently.

6.5.1 Duplicate values and secondary indexes

Again, duplicate keys pose a problem to secondary indexes. Think in the following setup:

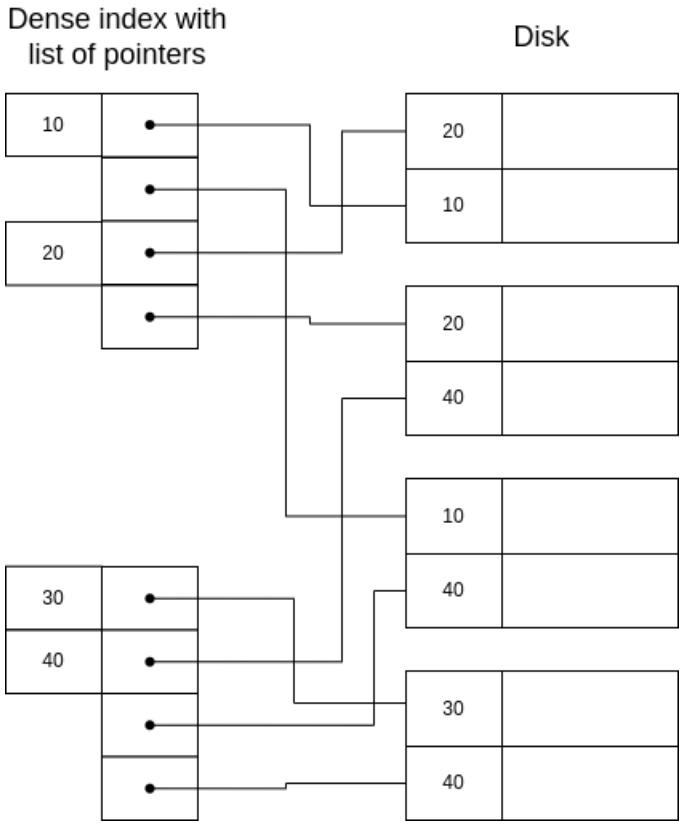


Here, the solutions that we proposed before don't work, because for them we needed sequentially stored files. In this case, again, the naïve solution is a dense index:

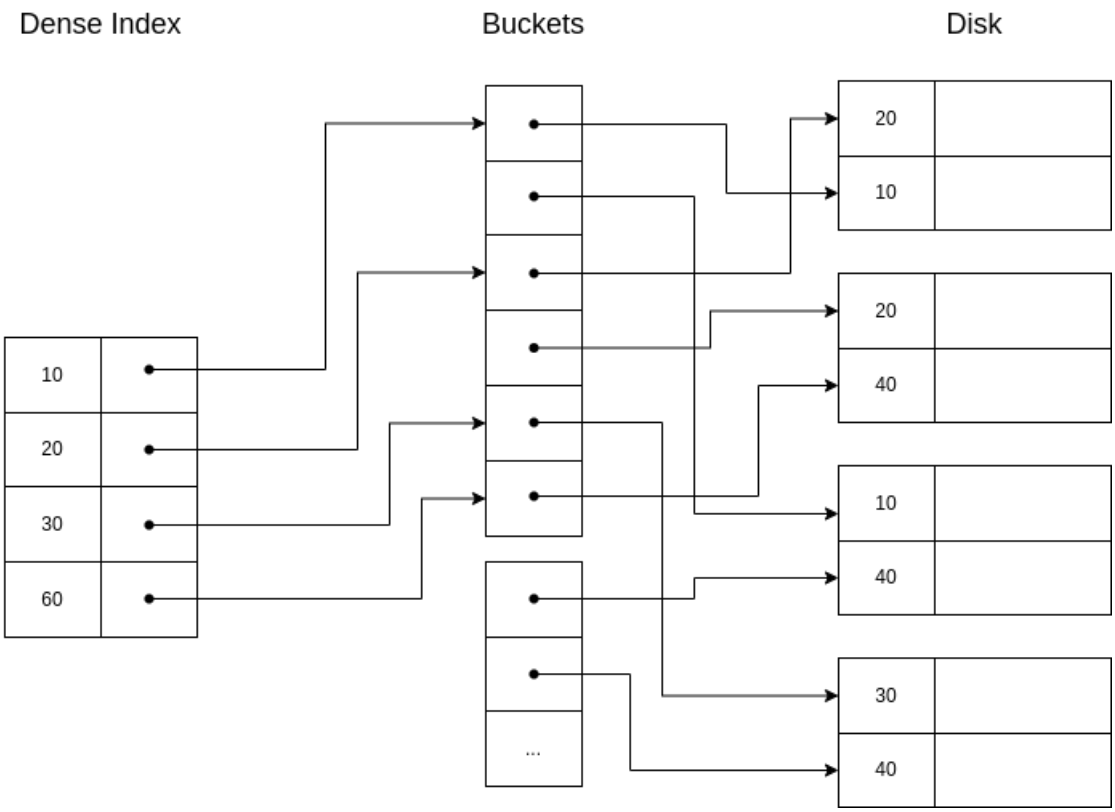


The problem with this solution is that this cause an excessive overhead, both in disk space (we are storing

repeatedly the same keys) and in search time (because the index keyset needs to be accessed several times per key). An alternative is to store only once each key, and associate a list of pointers:



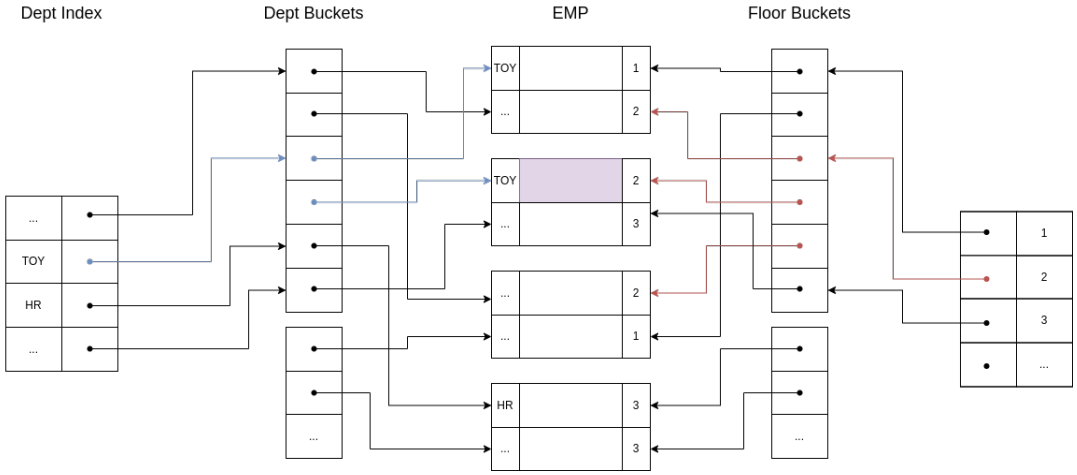
But this has the problem that the index entries can have different sizes, which difficult the search. Another idea is to use buckets of pointers:



This structure is very helpful in some situations, for example when we want to get data with some conditions that involves different fields indexed.

Example 6.11. Imagine the relation EMP(name,dept,floor), with a primary index on name and two secondary indexes with bucket structure in dept and floor.

Now, let's say we want to retrieve all employees in the department 'Toy' and in floor 2. Our structure make this query very easy:



As we can see, it is possible to use both indexes and then fetch only those records that are return by the two of them!

7 B-Trees

This section is adapted from [2].

B-trees automatically maintain as many levels of index as is appropriate for the size of the file being indexed and manage the space on the blocks they use so that every block is between half used and completely full.

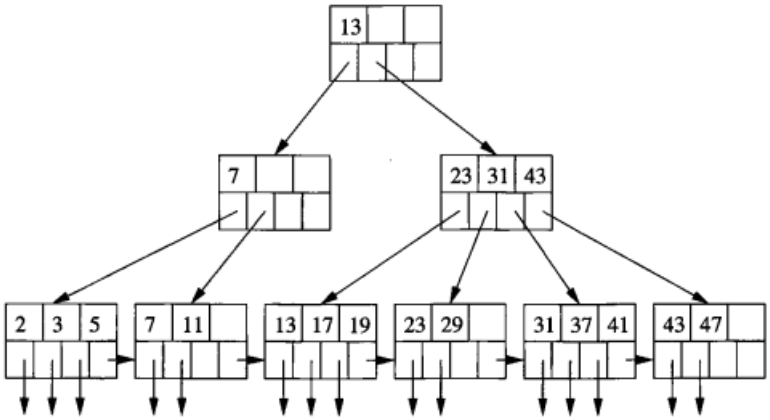


Figure 4: A BTree. Source: [2].

A B-tree organizes its blocks into a tree that is balanced, meaning that all paths from the root to a leaf have the same length. A BTree can be visualized in Figure 4.

There is a parameter n associated with each B-tree index, and this parameter determines the layout of all blocks of the B-tree. Each block will have space for n search-key values and $n+1$ pointers.

We pick n to be as large as will allow $n+1$ pointers and n keys to fit in one block.

Example 7.1. Suppose our blocks are 4096 bytes. Also let keys be integers of 4 bytes and let pointers be 8 bytes. If there is no header information kept on the blocks, then we want to find the largest integer value of n such that $4n + 8(n+1) < 4096$. That value is $n = 340$.

There are several important rules about what can appear in the blocks of a B-tree:

- The keys in leaf nodes are copies of keys from the data file. These keys are distributed among the leaves in sorted order, from left to right.
- At the root, there are at least two used pointers. All pointers point to B-tree blocks at the level below.
- At a leaf, the last pointer points to the next leaf block to the right, i.e., to the block with the next higher keys. Among the other n pointers in a leaf block, at least $\lfloor \frac{n+1}{2} \rfloor$ of these pointers are used and point to data records; unused pointers are null and do not point anywhere. The i^{th} pointer, if it is used, points to a record with the i^{th} key.
- At an interior node, all $n+1$ pointers can be used to point to B-tree blocks at the next lower level. At least $\lceil \frac{n+1}{2} \rceil$ of them are actually used (but if the node is the root, then we require only that at least 2 be used, regardless of how large n is). If j pointers are used, then there will be j—1 keys, say $K_1, K_2, ..., K_{j-1}$. The first pointer points to a part of the B-tree where some of the records with keys less than K_1 will be found. The second pointer goes to that part of the tree where all records with keys that are at least K_1, but less than K_2 will be found, and so on. Finally, the j^{th} pointer gets us to the part of the B-tree where some of the records with keys greater than or equal to K_{j-1} are found. Note that some records with keys far below K_1 or far above K_{j-1} may not be reachable from this block at all, but will be reached via another block at the same level.
- All used pointers and their keys appear at the beginning of the block, with the exception of the $(n+1)^{th}$ pointer in a leaf, which points to the next leaf.

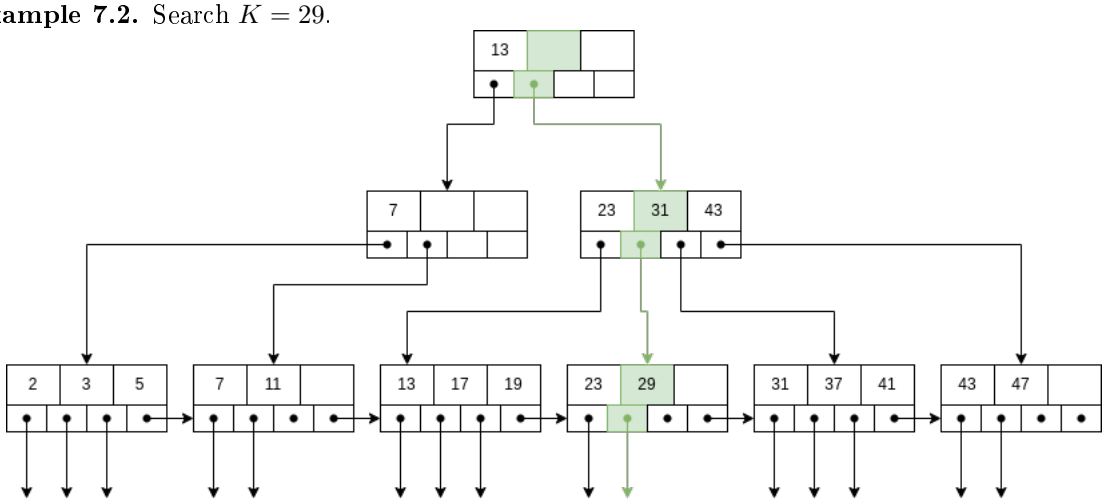
In Figure 4, the chosen n is 3.

7.1 Lookup in BTree

Suppose we want to find a record with key K. The procedure is:

- **Base case**: if we are at a leaf node, look among the keys. If the i^{th} key is K, then the i^{th} pointer is the one that we were looking for.
- **Inductive case**: if we are at an interior node with keys $K_1, ..., K_n$ we find i such that $K_{i-1} \leq K < K_n$, or 1 if $K < K_1$. We choose the i^{th} pointer to continue the search.

Example 7.2. Search $K = 29$.



The path to be followed is colored green:

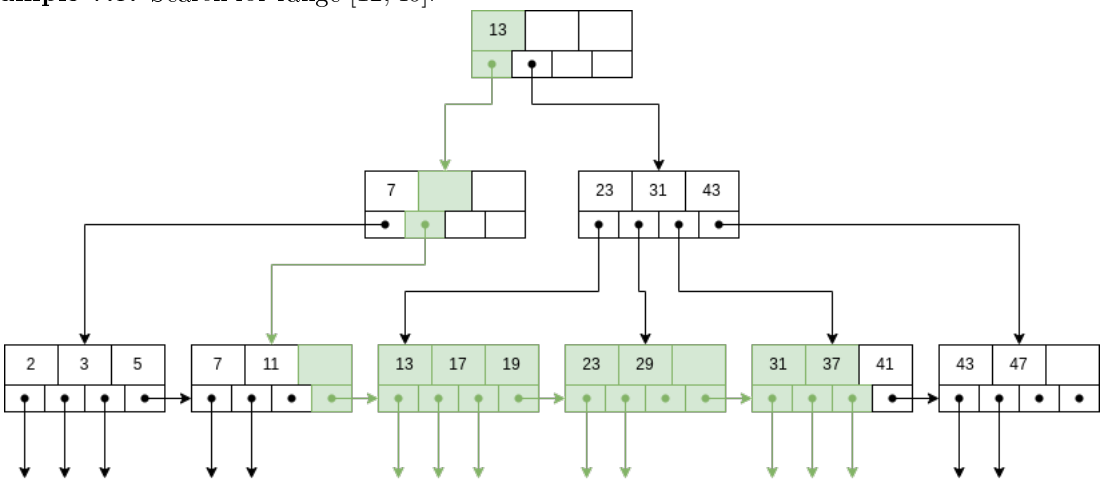
1. In the root node, $K_1 < K$ and and there are no more keys, so choose $i = 2$.
2. In the next node, we find $K_1 < K < K_2$, so choose $i = 2$.
3. In the leave node, we find $K_2 = K$, so choose $i = 2$.

7.2 Range queries

BTrees allow for a very efficient way to process range queries, that is, recover all records with keys lying in a given range $[K_{min}, K_{max}]$. The procedure is:

1. Perform a lookup for K_{min}, whether it is found or not, we will reach the correct leaf node.
2. We traverse all leaf nodes until we find a key bigger than K_{max}.

Example 7.3. Search for range $[12, 40]$.



It is colored in green all the nodes that would be accepted into the range query. The order is up-down and left-right in the leaf level.

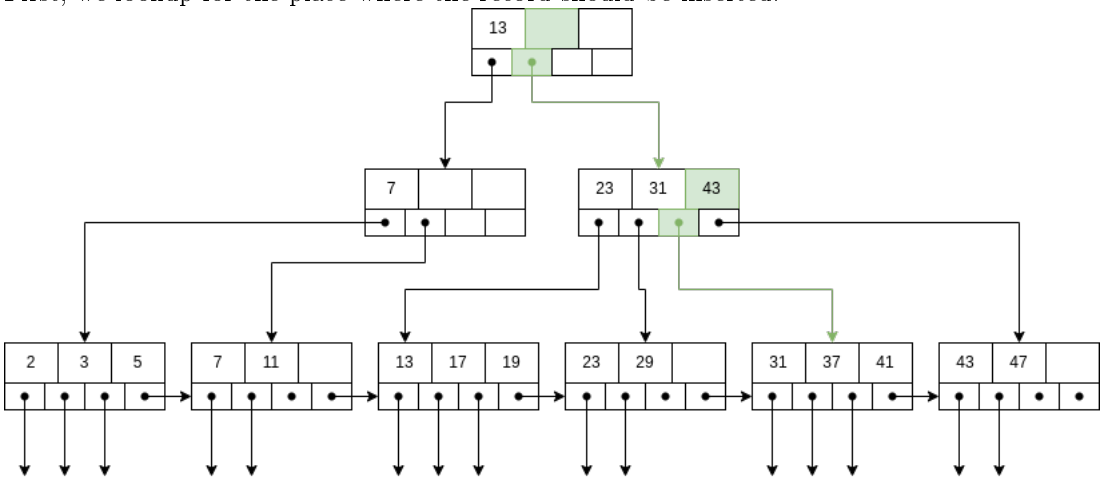
7.3 Insertion into a BTree

The insertion is, in principle, recursive:

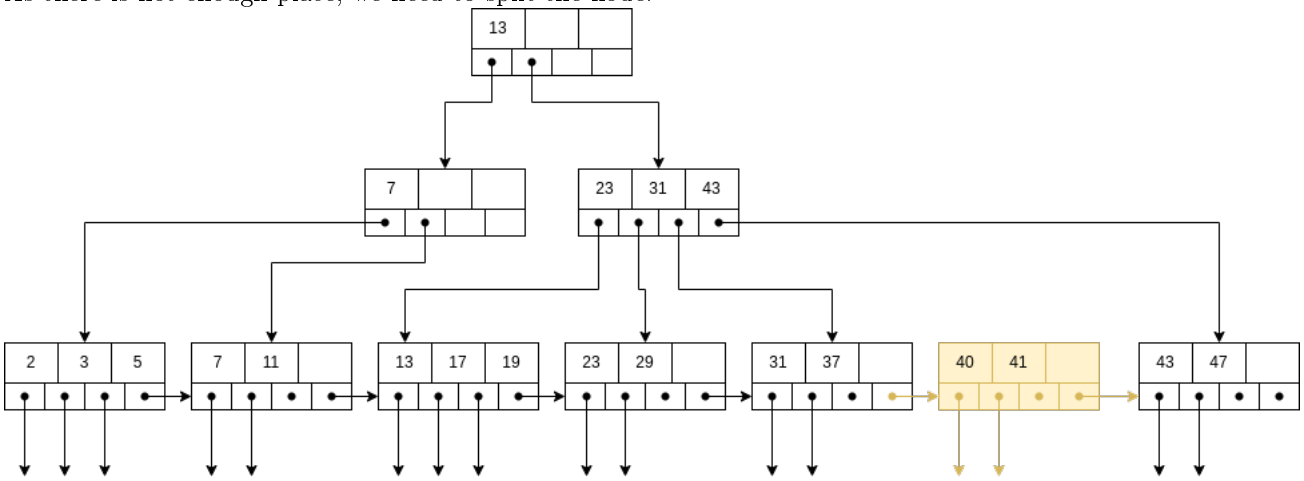
- We try to find a place for the new key in the appropriate leaf, and we put it there if there is room.
- If there is no room in the proper leaf, we split the leaf into two and divide the keys between the two new nodes, so each is half full or just over half full.
- The splitting of nodes at one level appears to the level above as if a new key-pointer pair needs to be inserted at that higher level. We may thus recursively apply this strategy to insert at the next level: if there is room, insert it; if not, split the parent node and continue up the tree.
- As an exception, if we try to insert into the root, and there is no room, then we split the root into two nodes and create a new root at the next higher level; the new root has the two nodes resulting from the split as its children. Recall that no matter how large n (the number of slots for keys at a node) is, it is always permissible for the root to have only one key and two children.

Example 7.4. Insert $K = 40$.

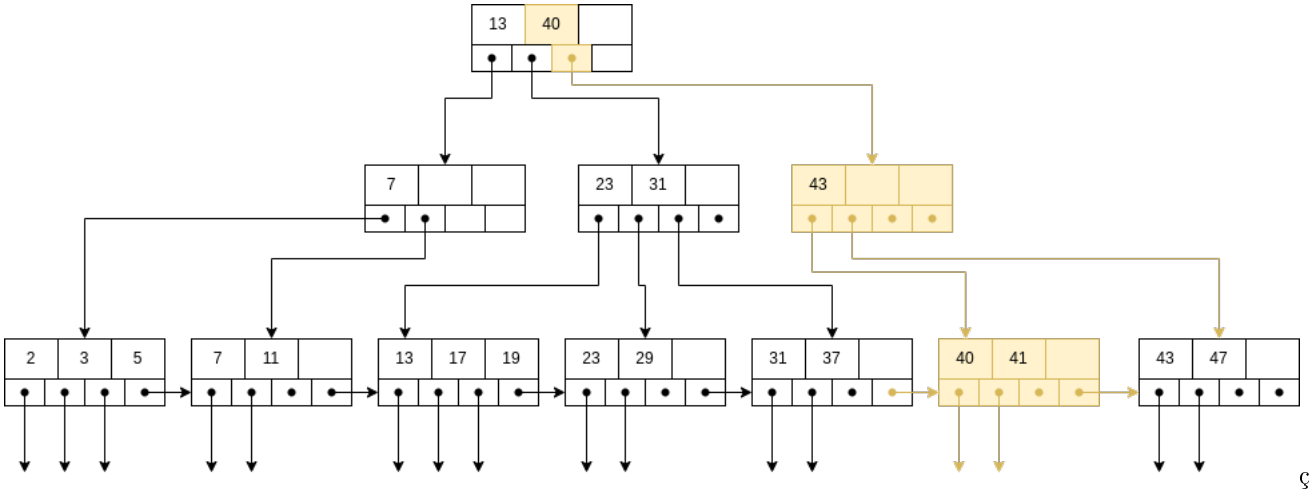
First, we lookup for the place where the record should be inserted.



As there is not enough place, we need to split the node.



Now, the key of the new node needs to be inserted into the parent node. But it is also full, so it needs to be splitted, too.



ç

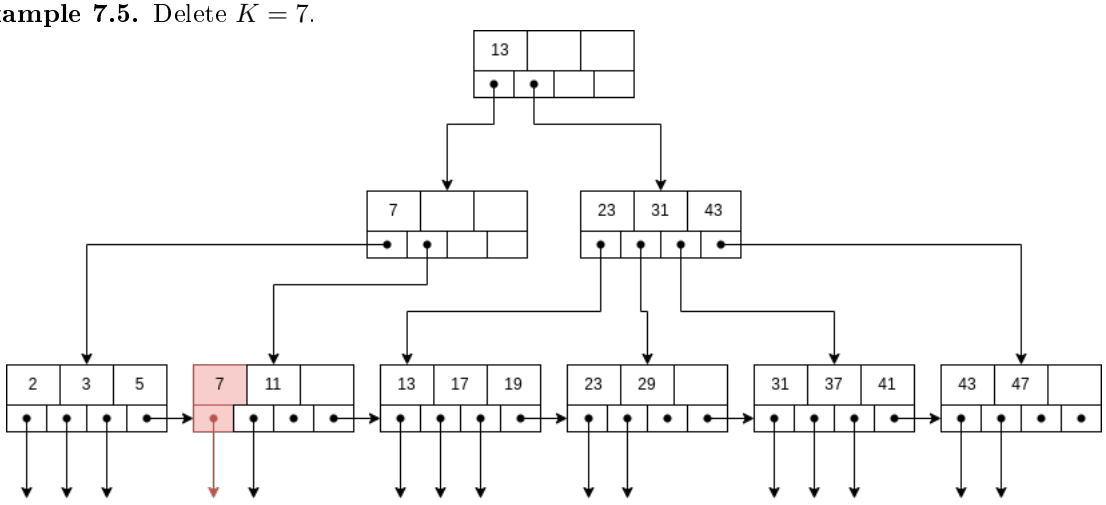
In the root node, the smaller key reachable from that node needs to bu inserted. In this case, the newly inserted one.

7.4 Deletion from a BTree

The steps to delete record with key K are:

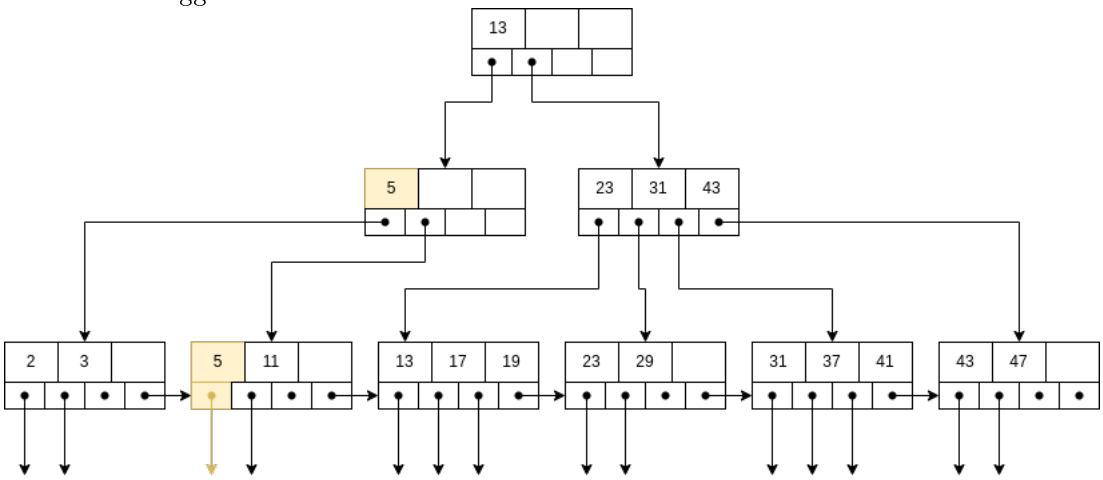
1. Lookup for the record.
2. Delete the record from the data.
3. Delete the key-pointer pair from the BTree.
4. If the node from which we deleted still has the minimum number of pointers, that's it. But it is possible that the node is less occupy than the minimum required after the deletion. We need to do one of two things:
   (a) If one of the adjacent siblings of node N has more than the minimum number of keys and pointers, then one key-pointer pair can be moved to N, keeping the order of keys intact. Possibly, the keys at the parent of N must be adjusted to reflect the new situation.
   (b) The hard case is when neither adjacent sibling can be used to provide an extra key for N. However, in that case, we have two adjacent nodes, N and a sibling M; the latter has the minimum number of keys and the former has fewer than the minimum. Therefore, together they have no more keys and pointers than are allowed in a single node. We merge these two nodes, effectively deleting one of them. We need to adjust the keys at the parent, and then delete a key and pointer at the parent. If the parent is still full enough, then we are done. If not, then we recursively apply the deletion algorithm at the parent. This process is called **coalesce siblings**.

Example 7.5. Delete $K = 7$.



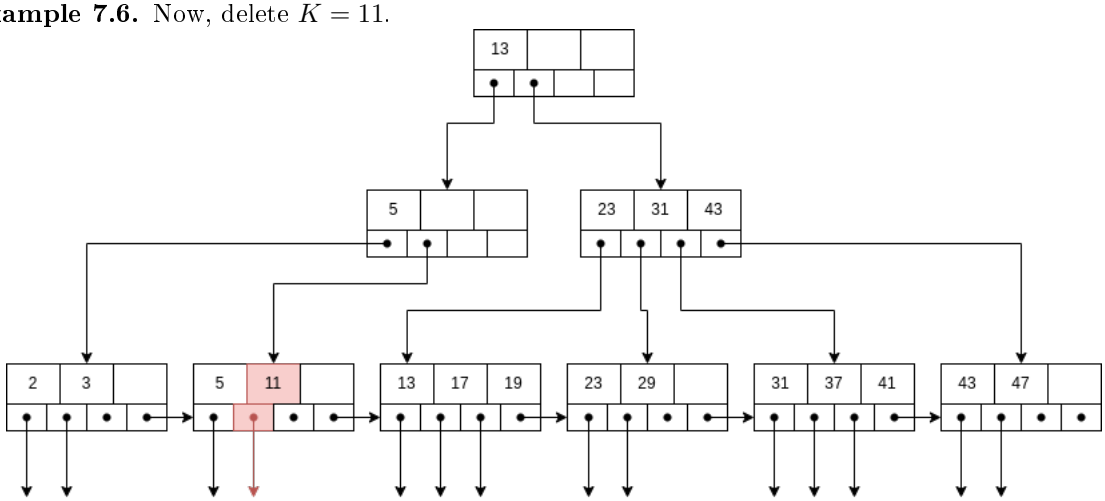
First, we find the correct node and delete the record.

Now, the node left only has one pointer, so we need to fix this. As its left sibling node has 3 pointers, we can transfer the biggest one.

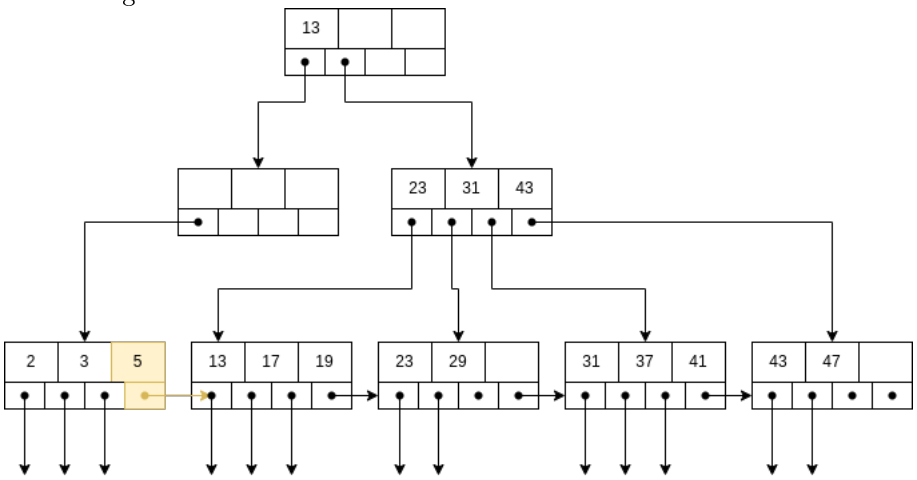


Also, we need to update the parent node.

Example 7.6. Now, delete $K = 11$.



First, we locate the correct node and delete the record. Again, the node ends up with less pointers than it should, but now the left sibling does not have more than the minimum amount of pointers (it has the minimum) and the node does not have right siblings (the node to the right is from another parent), so we need to merge the two siblings.



Part IV
Physical Query Plans

8 Physical Query Plans

We saw the steps to process a query: first, the query needs to be translated into and RA expression, which can then be modified using equivalence rules to get different expressions that lead to the same result. Then, it is needed to estimate the cost of each expression and to take the one that gives the minimum expected cost. For this, we need also to decide between several ways to access the data, e.g., whether to use an index or not, to order the data or not,... This is called **physical query planning**.

There are several ways to measure cost, but we are going to be using the number of disk blocks that must be read or written to execute a query plan.

We will also use different parameters:

- $B(R)$: number of blocks containing the tuples of relation R.
- $f(R)$: maximum number of tuples of R per block.
- $S(R)$: size of tuples of R.
- M: memory blocks available.
- $HT(i)$: amount of levels in index i.
- $LB(i)$: amount of leaf nodes in index i.

8.1 Computing joins

A join operation can be computed in several ways, depending on the options available.

The simplest and most costly option is an **iteration join**, which just performs a double loop over the two relations. The pseudocode can be read in Algorithm 2.

Algorithm 2 Iteration Join

```
for each r in R1 do
        for each s in R2 do
                if r.A = s.A then
                        output (r,s)
```

The **merge join** consists in first sorting the relations if they are not sorted, and then scanning them making use of the fact that they are ordered using the same attribute. The pseudocode can be read in Algorithm 3.

Algorithm 3 Merge Join

```
if R1 not sorted on attribute A then
        sort R1
if r2 not sorted on attribute A then
        sort R2

i=1, j=1
while i<=T(R1) and j<=T(R2) do
        if R1[i].A = R2[j].A then
                k=j
                while R1[i].A = R2[k].A do
                        output (R1[i],R2[k])
                        k += 1
                i += 1
        else if R1[i].A > R2[j].A then
                j += 1
        else if R1[i].A < R2[j].A then
                i += 1
```

The **index join** uses an index defined on the joining attribute on one of the relations. The pseudocode can be read in Algorithm 4.

Algorithm 4 Index Join

```
for each r in R1 do
        X <- index(R2, A r.A) # search in index on R2.A tuples with value r.A
        for each s in X do
                output (r,s)
```

The **hash join** uses a hash function on the joinin attribute. The pseudocode can be read in Algorithm

Algorithm 5 Hash Join (k buckets G1...Gk, H1...Hk)

```
hash R1 tuples into G buckets
hash R2 tuples into H buckets

for i=0 to k
        match tuples that lie in G[i] and H[i]
```

8.2 Factors that affect performance

Are the tuples of the relation stored physically together?

The more compactly stored in memory the relations are, the less number of pages needs to be fetched and so the performance will increase.

Are relations sorted by join attribute?

If the relations are already sorted by the join attribute, the merge join is a great option, because the costly part is the sorting ($O(n \log n)$), while the joinin itself is $O(n+m)$ where n, m are the sizes of both relations.

Indexes exist?

If there are no indexes, the index join is not even an option. And when there are indexes, they are not always the best option, because if the attribute has low selectivity, the indexes will be returning single values often,

and thus we will be only introducing overhead in the operation.

Example 8.1. Iteration join $R_1 \bowtie R_2$ where **relations are not contiguous**, $T(R_1) = 10000\, tuples, T(R_2) = 5000\, tuples, S(R_1) = S(R_2) = \frac{1}{10}\, block$ and $M = 101$ blocks (so we can work with at most 1010 tuples in memory at once). In this case, $B(R_1) = 10000$ and $B(R_2) = 5000$.

For each tuple in R_1, we need to:

- Read the tuple: 1 IO
- Read all tuples in R_2: 5000 IOs (This is because the relations are not contiguously stored, so we need to assume that we read a new block for each new tuple)

So, the total cost is

$$C = 10000\,(1 + 5000) = 50\ 010\ 000\ IOs.$$

Can this be improved? Yes!

If we do it reading 1000 tuples of R_1 and doing the process in each of this chunks, we would need to, for each chunk in $R1$:

- Read all tuples in the chunk: 1000 IOs
- Read all tuples in R_2: 5000 IOs

So

$$C = 10\,(1000 + 5000) = 60\ 000\ IOs.$$

Can this be improved? Yes!

If we reverse the order of the join: $R_2 \bowtie R_1$, then, for each chunk in $R2$:

- Read all tuples in the chunk: 1000 IOs
- Read all tuples in R_1:10000 IOs

So

$$C = 5\,(1000 + 10000) = 55\ 000\ IOs.$$

In fact, the bigger R_2 is compared to R_1 the greater gain obtained when changing the order.

Example 8.2. Iteration join $R_1 \bowtie R_2$ where **relations are contiguous** (same parameters). In this case, $B(R_1) = 1000$ and $B(R_2) = 500$.

For each chunk in R_2:

- Read the chunk: 100 IOs (now it is only 100 IOs because $S(R_2) = \frac{1}{10}$ block, so to read 1000 tuples we need to read 100 blocks)
- Read $R1$: 1000 IOs

Thus,

$$C = 5\,(100 + 1000) = 5\ 500\ IOs.$$

We can see how the contigous storage greatly increase performance of the joins.

Theorem 8.1. *In general, for an iteration join $R_1 \bowtie R_2$ with sizes (in blocks) $B(R_1)$ and $B(R_2)$, and a memory capacity of M blocks, the formula for the cost is*

$$C = \frac{B(R_1)}{M-1}\left(M - 1 + B(R_2)\right).$$

Proof. We want to first take as many blocks from the first relation as we can, but we need to leave space for joining the second relation, so we will use $M-1$ blocks for storing the tuples for the first relation. This will need to be done $\frac{B(R_1)}{M-1}$ times. Now, for each of this iterations, we need to actually read the $M-1$ blocks from R_1 and to read all blocks from R_2. So the formula arises. □

Example 8.3. Merge join $R_1 \bowtie R_2$ where both relations are **already ordered** by the joinin attribute and relations are **contiguous**. In this case, we will need to read all blocks containing R_1 and all block containing R_2, once. So

$$C = 1000 + 500 = 1\ 500\ IOs.$$

So we can see how good a merge join where the relations are already ordered is. Let's see how the other case performs:

Example 8.4. Merge join $R_1 \bowtie R_2$ where R_1, R_2 are **not ordered**, but are **contiguous**.

In this case, first we need to sort the relations and there are different ways to do this, we are going to explain one, the **merge join**:

- Merge sort: For each 100 tuples chunk of R:
  - Read the chunk
  - Sort in memory
  - Write to disk the ordered chunks
  - Read the ordered chunks and merge them
  - Write to disk the ordered relation

  Cost (in terms of IO): each chunk is read, written, read, written, so $4 \times \frac{T(R)}{S(R)} = 4B(R)$.

  In our case

$$sortC(R1) = 4000,\ sortC(R2) = 2000.$$

Now, the cost a merge join in which the relations are not ordered is the cost of the ordering plus the cost of the join, so

$$C = 4000 + 2000 + 1500 = 7500\ IOs.$$

Remember that the iteration cost was 5500 IOs, so in this case the merge join is not the best option.

Theorem 8.2. *In general, for a merge join $R_1 \bowtie R_2$ with sizes (in blocks) $B(R_1)$ and $B(R_2)$ where the relations are contiguously stored, and a memory capacity of M blocks, the formula for the cost is*

$$C = 5\left(B(R_1) + B(R_2)\right).$$

Proof. The cost is

$$C = C_{order}(R_1) + C_{order}(R_2) + C_{join}.$$

We have seen that $C_{order}(R) = 4B(R)$ and $C_{join}(R_1, R_2) = B(R_1) + B(R_2)$, and so the formula arises. □

Example 8.5. Let in this case R_1, R_2 be contiguously stored, but unordered, with $B(R_1) = 10000$ *tuples* and $B(R_2) = 5000$ *tuples*. In this case, the iteration join has a cost of

$$C_{IJ} = \frac{5000}{100}(100 + 10000) = 505\ 000\ IOs.$$

And the merge join

$$C_{MJ} = 5(10000 + 5000) = 75\ 000\ IOs.$$

So, in this case the merge sort is better, even without the relations being previously ordered.

Theorem 8.3. *For a join $R_1 \bowtie R_2$ where the relations are contiguously stored and unordered, with sizes (in blocks) $B(R_1)$ and $B(R_2)$, a memory capacity of M blocks, and assuming that $\frac{B(R_2)}{M-1} > 4$, a merge join is preferred to an iteration join if, and only if,*

$$B(R_1) > \frac{5B(R_2)}{\frac{B(R_2)}{M-1} - 4}.$$

Proof. The merge join is preferred to the iteration join if, and only if

$$5\left(B\left(R_1\right)+B\left(R_2\right)\right)<\frac{B\left(R_1\right)}{M-1}\left(M-1+B\left(R_2\right)\right)=B\left(R_1\right)+\frac{B\left(R_1\right)B\left(R_2\right)}{M-1} \iff$$

$$4B\left(R_1\right)+5B\left(R_2\right)<\frac{B\left(R_1\right)B\left(R_2\right)}{M-1} \iff$$

$$B\left(R_1\right)\left[4-\frac{B\left(R_2\right)}{M-1}\right]<-5B\left(R_2\right) \iff$$

$$B\left(R_1\right)\left[\frac{B\left(R_2\right)}{M-1}-4\right]>5B\left(R_2\right) \iff$$

$$B\left(R_1\right)>\frac{5B\left(R_2\right)}{\frac{B(R_2)}{M-1}-4}.$$

□

Example 8.6. Let's apply the theorem to our two previous examples:

1. $R_2 \bowtie R_1, B\left(R_1\right)=1000,\ B\left(R_2\right)=500,\ M=101$, then

$$\frac{5B\left(R_1\right)}{\frac{B(R_1)}{M-1}-4}=\frac{5000}{\frac{1000}{100}-4}=\frac{5000}{6}>833>500,$$

   so in this case the iteration join is preferred.

2. $R_2 \bowtie R_1, B\left(R_1\right)=10000, B\left(R_2\right)=5000, M=101$, then

$$\frac{5B\left(R_1\right)}{\frac{B(R_1)}{M-1}-4}=\frac{50000}{\frac{10000}{100}-4}=\frac{50000}{96}<521<5000,$$

   so in this case the merge join is preferred.

How much memory do we need for merge sort?

Until now, we have disregarded the memory needed to perform the merge sort, but this is a crucial aspect of it. If the relation does not fit entirely in memory, it is not straightforward to merge all the ordered chunks to obtain a fully ordered relation.

In general, if we have M blocks in memory, and B blocks to sort, then we will take chunks of size k, so we will have $\frac{x}{k}$ chunks. Now, this number needs to be smaller than the memory size:

$$\frac{B}{M} \leq M,$$

or, equivalently

$$M^2 \geq B \text{ or } M \geq \sqrt{B}.$$

Example 8.7. Following our examples: R_1 is 1000 blocks, so $M \geq 31.62$ and R_2 is 500 blocks so $M \geq 22.36$. In this case, we need $M \geq 32$ blocks, so it could be done because the memory was $M = 101$ blocks.

Can the merge join be improved?

Yes, we are imposing that the whole relation needs to be sorted, but maybe we can join the sorted chunks without merging them.

If we did this, we would need to:

- Read R_1 and write R_1 sorted chunks
- Read R_2 and write R_2 sorted chunks
- Join

So the total cost would be

$$C = 2B(R_1) + 2B(R_2) + [B(R_1) + B(R_2)] = 3[B(R_1) + B(R_2)].$$

Example 8.8. Index join $R_2 \bowtie R_1$ with an **index** on $R_1.A$ of two levels, R_2 **contiguously** stored and **unordered** and assuming the **index fits in memory**. Then, the cost is:

- Read R_2: 500 IOs
- For each tuple in R_2, check the index and only read a tuple in R_1 if there is a match.

So

$$C = 500 + matches.$$

Thus, we need to estimate how many matches there will be. We can treat several cases:

1. If $R_1.A$ is a key attribute and $R_2.A$ is a foreign key:

   $$matches(R_2, R_1) = T(R_2).$$

   In this case the cost is

   $$C = 500 + 5000 = 5500\ IOs.$$

2. If we know $V(R_1, A)$ (number of distinct values of attribute A in R_1) and $T(R_1)$, we can assume uniformity and thus obtain

   $$matches(R_2, R_1) = \frac{V(R_1, A)}{T(R_1)} \times T(R_2).$$

   In this case the cost is, assuming $V(R_1, A) = 5000$,

   $$C = 500 + \frac{10000}{5000}5000 = 10500\ IOs.$$

3. If we know $size(Dom(R_1, A))$ (number of distinct values that attribute A can take) and $T(R_1)$, we can assume uniformity and thus obtain

   $$matches(R_2, R_1) = \frac{T(R_1)}{size(Dom(R_1, A))}T(R_2).$$

   In this case the cost is, assumiing $size(Dom(R_1, A)) = 1\ 000\ 000$,

   $$C = 500 + \frac{10000}{1000000}5000 = 550\ IOs.$$

Example 8.9. Let's see what happens if the index does not fit in memory.

Let the $R_1.A$ index occupy 201 blocks (1 root and 200 leaves), so it cannot be fully fitted in memory ($M = 101$). We can store the root node and 99 leaf nodes in memory. Then for each value to check, there is a $\frac{99}{200}$ chance that we can find the value in memory, and $\frac{101}{200}$ that we don't. Then, the cost of checking the value in the index is

$$C_{index} = 0 \times \frac{99}{200} + 1 \times \frac{101}{200} \approx 0.5.$$

Thus, the total cost is

$$C = 500 + 5000[0.5 + 2] = 13000\ IOs.$$

In this case, we have assumed the case 2. The detailed explanation is:

- Read R_2: 500 IOs
- For each tuple in R_2 (5000×):
  - We need to check the value in the index: 0.5 IOs
  - And we need to recover the matches: assuming case 2 is 2 IOs, assuming case 3 is 0.1 IOs

Theorem 8.4. *In general, for an index join $R_1 \bowtie_A R_2$ with sizes (in blocks) $B(R_1)$ and $B(R_2)$ and sizes (in tuples) $T(R_1)$ and $T(R_2)$, where:*

- *There is an index for $R_1.A$ of size B_i.*
- *R_2 is contiguously stored in memory.*
- *There is a memory capacity of M.*

If the index does not fit in memory, the estimated cost is

$$C = B(R_2) + T(R_2)\left[\left(1 - \frac{M-2}{S_i}\right) + matched\right],$$

where matched depends on the assumtions about how the values of the index are distributed.
If the index fits in memory, the estimated cost is

$$C = B(R_2) + T(R_2) \times matched.$$

Proof. We will need to read all blocks of R_2, which sums up to $B(R_2)$.

Then, for each tuple in R_2, we need to check the index and recover all matches. Thus, if the index fits in memory, the result is obvious.

If the index does not fit in memory, we will store the root and $M-2$ leave nodes. Thus

$$Prob(value\ in\ memory) = \frac{M-2}{S_i},$$

so

$$Prob(value\ not\ in\ memory) = 1 - Prob(value\ in\ memory) = 1 - \frac{M-2}{S_i}.$$

And so, we obtain the desired formula. □

Let's now continue with the hash join:

Example 8.10. Hash join $R_1 \bowtie R_2$ where R_1, R_2 are **contiguously stored** and **unordered**. According to [2], we may hash each relation to 100 buckets, so the average size of a bucket is 10 blocks for R_1 and 5 blocks for R_2. Since the smaller number, 5, is much less than the number of available buffers, we expect to have no trouble performing a one-pass join on each pair of buckets. The number of disk IOs is 1500 to read each of R_1 and R_2 while hashing into buckets, another 1500 to write all the buckets to disk, and a third 1500 to read each pair of buckets into main memory again while taking the one-pass join of corresponding buckets. Thus, the total cost is

$$C = 3(B(R_1) + B(R_2)) = 4500\ IOs.$$

About the memory requirements, we need the buckets to fit into memory. We are taking $M-1$ buckets, so the size of the buckets are $\frac{B(R_1)}{M-1}$ blocks and $\frac{B(R_2)}{M-1}$ blocks for buckets of R_1 and buckets of R_2, respectively. It is enough to fit the smaller one, say $\frac{B}{M-1}$ blocks. Then, we need to ensure that

$$\frac{B}{M-1} < M-1,$$

so we need to fulfill

$$B < (M-1)^2$$

or

$$\sqrt{B} < M-1.$$

Part V
Extensibility

9 Extensible databases: PostgreSQL

This section is adapted from the course slides and PostgreSQL: Extensibility.

PostgreSQL is extensible because its operation is catalog-driven. Relational DB systems store information about databases, tables, columns, etc., in what are commonly known as system catalogs.

The catalogs 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 catalogs: 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.

The PostgreSQL server can moreover incorporate user-written code into itself through dynamic loading. That is, the user can specify an object code file (e.g., a shared library) that implements a new type or function, and PostgreSQL will load it as required. Code written in SQL is even more trivial to add to the server. This ability to modify its operation "on the fly" makes PostgreSQL uniquely suited for rapid prototyping of new applications and storage structures.

9.1 Types

9.1.1 Base types

Base types are those, like integer, that are implemented below the level of the SQL language. PostgreSQL can only operate on such types through functions provided by the user and only understands the behavior of such types to the extent that the user describes them.

9.1.2 Container types

Container types can be arrays, composites and ranges:

- **Arrays** can hold multiple values that are all of the same type. An array type is automatically created for each base type, composite type, range type, and domain type. But there are no arrays of arrays.
- **Composite** types, or row types, are created whenever the user creates a table. It is also possible to use CREATE TYPE to define a "stand-alone" composite type with no associated table. A composite type is simply a list of types with associated field names. A value of a composite type is a row or record of field values.
- A **range** type can hold two values of the same type, which are the lower and upper bounds of the range. Range types are user-created, although a few built-in ones exist.

9.1.3 Domains

A domain is based on a particular underlying type and for many purposes is interchangeable with its underlying type. However, a domain can have constraints that restrict its valid values to a subset of what the underlying type would allow. Domains are created using the SQL command CREATE DOMAIN.

9.1.4 Pseudo-types

There are a few "pseudo-types" for special purposes. Pseudo-types cannot appear as columns of tables or components of container types, but they can be used to declare the argument and result types of functions. This provides a mechanism within the type system to identify special classes of functions.

9.1.5 Polymorphic types

Some pseudo-types of special interest are the polymorphic types, which are used to declare polymorphic functions. This powerful feature allows a single function definition to operate on many different data types, with the specific data type(s) being determined by the data types actually passed to it in a particular call.

9.2 Functions

PostgreSQL provides four kinds of functions:

- query language functions (functions written in SQL)
- procedural language functions (functions written in, for example, PL/pgSQL or PL/Tcl)
- internal functions
- C-language functions

Every kind of function can take base types, composite types, or combinations of these as arguments (parameters). In addition, every kind of function can return a base type or a composite type. Functions can also be defined to return sets of base or composite values.

9.2.1 SQL functions

SQL functions execute an arbitrary list of SQL statements, returning the result of the last query in the list. In the simple (non-set) case, the first row of the last query's result will be returned. (Bear in mind that "the first row" of a multirow result is not well-defined unless you use ORDER BY). If the last query happens to return no rows at all, the null value will be returned.

Alternatively, an SQL function can be declared to return a set (that is, multiple rows) by specifying the function's return type as SETOF sometype, or equivalently by declaring it as RETURNS TABLE(columns). In this case all rows of the last query's result are returned.

Any collection of commands in the SQL language can be packaged together and defined as a function. However, the final command must be a SELECT or have a RETURNING clause that returns whatever is specified as the function's return type. Alternatively, if you want to define an SQL function that performs actions but has no useful value to return, you can define it as returning void.

Example 9.1. A SQL function defined by the user.

```
CREATE FUNCTION clean_emp() RETURNS void AS '
    DELETE FROM emp
        WHERE salary < 0;
' LANGUAGE SQL;
```

Remark 9.1. The entire body of an SQL function is parsed before any of it is executed. While an SQL function can contain commands that alter the system catalogs (e.g., CREATE TABLE), the effects of such commands will not be visible during parse analysis of later commands in the function. Thus, for example, CREATE TABLE foo (...); INSERT INTO foo VALUES(...); will not work as desired if packaged up into a single SQL function, since foo won't exist yet when the INSERT command is parsed. It's recommended to use PL/pgSQL instead of an SQL function in this type of situation.

Remark 9.2. More than one function can be defined with the same SQL name, so long as the arguments they take are different. In other words, function names can be overloaded. This is called **function overloading**.

9.2.2 Procedural functions

PostgreSQL allows user-defined functions to be written in other languages besides SQL and C. These other languages are generically called procedural languages (PLs). Procedural languages aren't built into the PostgreSQL server; they are offered by loadable modules.

9.2.3 Internal functions

Internal functions are functions written in C that have been statically linked into the PostgreSQL server. The "body" of the function definition specifies the C-language name of the function, which need not be the same as the name being declared for SQL use.

Normally, all internal functions present in the server are declared during the initialization of the database cluster, but a user could use CREATE FUNCTION to create additional alias names for an internal function. Internal functions are declared in CREATE FUNCTION with language name internal.

Example 9.2. An internal function.

```
CREATE FUNCTION square_root(double precision) RETURNS double precision
    AS 'dsqrt'
    LANGUAGE internal
    STRICT;
```

9.2.4 C-Language functions

User-defined functions can be written in C (or a language that can be made compatible with C, such as C++). Such functions are compiled into dynamically loadable objects (also called shared libraries) and are loaded by the server on demand. The dynamic loading feature is what distinguishes "C language" functions from "internal" functions — the actual coding conventions are essentially the same for both. (Hence, the standard internal function library is a rich source of coding examples for user-defined C functions.)

Currently only one calling convention is used for C functions ("version 1"). Support for that calling convention is indicated by writing a PG_FUNCTION_INFO_V1() macro call for the function.

9.2.5 Function volatility categories

Every function has a volatility classification, with the possibilities being VOLATILE, STABLE, or IMMUTABLE. VOLATILE is the default if the CREATE FUNCTION command does not specify a category. The volatility category is a promise to the optimizer about the behavior of the function:

- A **VOLATILE** function can do anything, including modifying the database. It can return different results on successive calls with the same arguments. The optimizer makes no assumptions about the behavior of such functions. A query using a volatile function will re-evaluate the function at every row where its value is needed.
- A **STABLE** function cannot modify the database and is guaranteed to return the same results given the same arguments for all rows within a single statement. This category allows the optimizer to optimize multiple calls of the function to a single call. In particular, it is safe to use an expression containing such a function in an index scan condition. (Since an index scan will evaluate the comparison value only once, not once at each row, it is not valid to use a VOLATILE function in an index scan condition.)
- An **IMMUTABLE** function cannot modify the database and is guaranteed to return the same results given the same arguments forever. This category allows the optimizer to pre-evaluate the function when a query calls it with constant arguments. For example, a query like SELECT ... WHERE x = 2 + 2 can be simplified on sight to SELECT ... WHERE x = 4, because the function underlying the integer addition operator is marked IMMUTABLE.

For best optimization results, you should label your functions with the strictest volatility category that is valid for them.

9.3 Procedures

A procedure is a database object similar to a function. The key differences are:

- Procedures are defined with the CREATE PROCEDURE command, not CREATE FUNCTION.
- Procedures do not return a function value; hence CREATE PROCEDURE lacks a RETURNS clause. However, procedures can instead return data to their callers via output parameters

- While a function is called as part of a query or DML (data manipulation language) command, a procedure is called in isolation using the CALL command.
- A procedure can commit or roll back transactions during its execution (then automatically beginning a new transaction), so long as the invoking CALL command is not part of an explicit transaction block. A function cannot do that.
- Certain function attributes, such as strictness, don't apply to procedures. Those attributes control how the function is used in a query, which isn't relevant to procedures.

Collectively, functions and procedures are also known as **routines**. There are commands such as ALTER ROUTINE and DROP ROUTINE that can operate on functions and procedures without having to know which kind it is. Note, however, that there is no CREATE ROUTINE command.

9.4 Interfacing extensions to indexes

The procedures described thus far let you define new types, new functions, and new operators. However, we cannot yet define an index on a column of a new data type. To do this, we must define an **operator class** for the new data type.Operator classes can be grouped into operator families to show the relationships between semantically compatible classes. When only a single data type is involved, an operator class is sufficient.

The operators associated with an operator class are identified by "strategy numbers", which serve to identify the semantics of each operator within the context of its operator class. For example, B-trees impose a strict ordering on keys, lesser to greater, and so operators like "less than" and "greater than or equal to" are interesting with respect to a B-tree. Because PostgreSQL allows the user to define operators, PostgreSQL cannot look at the name of an operator (e.g., $<$ or $>=$) and tell what kind of comparison it is. Instead, the index method defines a set of "strategies", which can be thought of as generalized operators. Each operator class specifies which actual operator corresponds to each strategy for a particular data type and interpretation of the index semantics.

Example 9.3. The B-tree index method defines five strategies, shown in the next Table.

Operation	Strategy Number
less than	1
less than or equal	2
equal	3
greater than or equal	4
greater than	5

9.5 Steps to create a PostgreSQL extension

1. Create the appropriate file structure: $extension-version.sql$, $extension.c$, $Makefile$, $extension.control$.
2. Create the data types.
3. Create I/O functions.
4. Create constructors, getters, setters.
5. Create needed functions.
6. Create operators $=, <, \leq, >, \geq, ...$
7. Define operator classes for indexes.

Part VI
Failure Recovery and concurrency control

10 Failure recovery

Definition 10.1. Integrity constraints are predicates that all data in the database must satisfy. A database is said to be in a **consistent state** if it satisfies all constraints defined on it. In such case, the database itself is said to be a **consistent DB**.

Remark 10.1. Databases cannot be consistent at all times, because when some operations are being done, it is possible to be in intermediate non-consistent states.

A **transaction** is a collection of actions that preserve consistency. Thus, a transaction should be the smallest unit of processing in the database.

A fundamental assumption about transactions is the correctness principle:

Correctness principle: If a transaction executes in the absence of any other transactions or system errors, and it starts with the database in a consistent state, then the database is also in a consistent state when the transaction ends.

There is a converse to the correctness principle that forms the motivation for both the logging techniques that we are going to see. This converse involves two points:

1. A transaction is **atomic**, that is, it must be executed as a whole or not at all. If only part of a transaction executes, then the resulting database state may not be consistent. For example, if the system crashes in the middle of a transaction, if there is a media failure,...
2. Transactions that execute simultaneously are likely to lead to an inconsistent state unless we take steps to control their interactions (refer to Section 11).

In order to study the details of logging algorithms and other transactionmanagement algorithms, we need a notation that describes all the operations that move data between address spaces. The primitives we shall use are:

- INPUT(X): Copy the disk block containing database element X to a memory buffer.
- READ(X,t): Copy the database element X to the transaction's local variable t. More precisely, if the block containing database element X is not in a memory buffer then first execute INPUT (X). Next, assign the value of X to local variable t.
- WRITE(X,t): Copy the value of local variable t to database element X in a memory buffer. More precisely, if the block containing database element X is not in a memory buffer then execute INPUT(X). Next, copy the value of t to X in the buffer.
- OUTPUT (X): Copy the block containing X from its buffer to disk

10.1 Key problem: unfinished transactions

Unfinished transactions are a great problem when dealing with consistency. If we assume the correctness principle and all transactions execute completely (and isolated) then databases would always be consistent, and we would not be studying this, so there are reasons that makes transactions not to finish completely, leading to inconsistent states.

Example 10.1. Imagine we impose the constraint $A = B$ and we want to execute the transaction

$$\begin{aligned} T_1 : & A \leftarrow A \times 2 \\ & B \leftarrow B \times 2 \end{aligned}$$

It is obvious that if the database is consistent at the beginning of the transaction, it will also be consistent at the end, because both values start being the same, and they are modified in the same way.

Let's see how things can go wrong.

Imagine the following plan:

$$\begin{array}{lll} T_1 : & Read\,(A,t)\,; & \\ & Write\,(A,t)\,; & t \leftarrow t \times 2 \\ & Read\,(B,t)\,; & \\ & Write\,(B,t)\,; & t \leftarrow t \times 2 \\ & Output\,(A)\,; & \\ & Output\,(B)\,; & \end{array}$$

The initial state is:

Memory		**Disk**	
		A	8
		B	8

After $Read\,(A,t)$:

Memory		**Disk**	
A	8	A	8
		B	8
t	8		

After $t \leftarrow t \times 2$:

Memory		**Disk**	
A	8	A	8
		B	8
t	16		

After $Write\,(A,t)$:

Memory		**Disk**	
A	16	A	8
		B	8
t	16		

After $Read\,(B,t)$:

Memory		**Disk**	
A	16	A	8
B	8	B	8
t	8		

After $t \leftarrow t \times 2$:

Memory		**Disk**	
A	16	A	8
B	8	B	8
t	16		

After $Write\,(B,t)$:

Memory		**Disk**	
A	16	A	8
B	16	B	8
t	16		

After $Output\,(A)$:

Memory		Disk	
A	16	A	16
B	16	B	8
t	16		

After *Output* (B):

Memory		Disk	
A	16	A	16
B	16	B	16
t	16		

If all actions execute, as we can see, the final state is consistent. Nonetheless, there is one point in the procedure when a failure in the system can leave it in an inconsistent state: after *Output* (A) and before *Output* (B) the database is inconsistent!

We need to be able to ensure **atomicity** of transactions: all actions are executed, or none of them. For this purpose, **logging** is an useful technique. Basically, the idea is to annotate all actions done in a file, and if the system crashes, we can consult this file and rollback unfinished transactions, continuing from the point left,...

10.2 Logging

Definition 10.2. A **log** is a file of log records, each telling something about what some transaction has done. If log records appear in nonvolatile storage, we can use them to restore the database to a consistent state after a system crash.

There are several forms of log record that are used with each of the types of logging we discuss in this chapter. These are:

1. <START T>: This record indicates that transaction T has begun.
2. <COMMIT T>: Transaction T has completed successfully and will make no more changes to database elements. Any changes to the database made by T should appear on disk. However, because we cannot control when the buffer manager chooses to copy blocks from memory to disk, we cannot in general be sure that the changes are already on disk when we see the <COMMIT T> log record. If we insist that the changes already be on disk, this requirement must be enforced by the log manager (as is the case for undo logging).
3. < ABORT T>: Transaction T could not complete successfully. If transaction T aborts, no changes it made can have been copied to disk, and it is the job of the transaction manager to make sure that such changes never appear on disk, or that their effect on disk is cancelled if they do.

10.2.1 Undo logging

Undo logging makes repairs to the database state by undoing the effects of transactions that may not have completed before the crash.

For an undo log, the only other kind of log record we need is an update record, which is a triple < T ,X ,v> . The meaning of this record is: transaction T has changed database element X , and its former value was v. The change reflected by an update record normally occurs in memory, not disk.

An undo log does not record the new value of a database element, only the old value. If recovery is necessary in a system using undo logging, the only thing the recovery manager will do is cancel the possible effect of a transaction on disk by restoring the old value.

Example 10.2. Let's repeat the previous example with an undo log added to the scheme.

Initially:

Memory		Disk		Log
		A	8	
		B	8	

The transaction T_1 starts:

Memory		Disk		Log
		A	8	<T1,start>
		B	8	

Read (A, t) ; $t \leftarrow t \times 2$:

Memory		Disk		Log
A	8	A	8	<T1,start>
		B	8	
t	16			

Write (A, t):

Memory		Disk		Log
A	16	A	8	<T1,start>
		B	8	<T1,A,8>
t	16			

Reat (B, t) ; $t \leftarrow t \times 2$:

Memory		Disk		Log
A	16	A	8	<T1,start>
B	8	B	8	<T1,A,8>
t	16			

Write (B, t):

Memory		Disk		Log
A	16	A	8	<T1,start>
B	16	B	8	<T1,A,8>
t	16			<T1,B,8>

Output (A):

Memory		Disk		Log
A	16	A	16	<T1,start>
B	16	B	8	<T1,A,8>
t	16			<T1,B,8>

Output (B):

Memory		Disk		Log
A	16	A	16	<T1,start>
B	16	B	16	<T1,A,8>
t	16			<T1,B,8>
				<T1,commit>

Imagine the system crashes after *Output* (A) and before *Output* (B). When we switch on the system again, the database system manager would check the log and see that T_1 is unfinished, so it would set the values of A and B to be 8, and consistency would be recovered.

Complications

Another aspect to take into account, is that the log must be first written in memory, not written to disk on every action, so some problems can arise.

Example 10.3. First bad state: DB modified before log is written

Memory	
A	16
B	16
t	16
Log:	
<T1,start>	
<T1,A,8>	
<T1,B,8>	

Disk	
A	16
B	8

Log

If the system crashes now, we lose the log information in memory, and we don't have it on disk, so we would not be able to recover to the previous consistent state.

Example 10.4. Second bad state: log written before DB modified

Memory	
A	16
B	16
t	16
Log:	
<T1,start>	
<T1,A,8>	
<T1,B,8>	
<T1,commit>	

Disk	
A	16
B	8

Log
<T1,start>
<T1,A,8>
<T1,B,8>
<T1,commit>

If the system fails now, we would think that B has been correctly chanegd, because the buffer manager has not issued the $Output\,(B)$ operator yet.

An undo log is sufficient to allow recovery from a system failure, provided transactions and the buffer manager obey two rules:

U1) If transaction T modifies database element X , then the log record of the form < T ,X ,v> must be written to disk before the new value of X is written to disk.

U2) If a transaction commits, then its COMMIT log record must be written to disk only after all database elements changed by the transaction have been written to disk, but as soon thereafter as possible.

In order to force log records to disk, the log manager needs a fiush-log command that tells the buffer manager to copy to disk any log blocks that have not previously been copied to disk or that have been changed since they were last copied. In sequences of actions, we shall show FLUSH LOG explicitly.

Example 10.5. Let's repeat the example with all this rules:

Step	Action	t	MemA	MemB	DiskA	DiskB	MemLog	DiskLog
1							<Start T>	
2	$Read(A,t)$	8	8		8	8		
3	$t \leftarrow t \times 2$	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	$t \leftarrow t \times 2$	16	16	8	8	8		
7	$Write\,(B,t)$	16	16	16	8	8	<T,B,8>	
8	$FlushLog$							<Start T>;<T,A,8>;<T,B,8>
9	$Output\,(A)$	16	16	16	16	8		
10	$Output\,(B)$	16	16	16	16	16		
11							<Commit T>	
12	$FlushLog$							<Commit T>

In this case, at any point in the process, if there is a failure, we would be able to rollback to a previous consistent state using the log written in disk.

When an error ocurrs, there is a procedure to follow, which is detailed in Algorithm 6. Note that if a failure occurs during recovery, nothing goes wrong, because the recovery procedure would start over again when the system is switched on and the rollback operations would proceed in the same manner.

Algorithm 6 Undo logging: recovery rules

```
Let S = set of transaction with <Ti, start> in log, but no
                <Ti, commit> not <Ti, abort> in log

for each <Ti,X,v> in log in reverse order do
        if Ti in S then
                write(X, v)
                output(X)

for each Ti in S do
        write <Ti, abort> to log
```

10.2.2 Redo logging

Undo logging has a potential problem that we cannot commit a transaction without first writing all its changed data to disk. Sometimes, we can save disk I/O 's if we let changes to the database reside only in main memory for a while. As long as there is a log to fix things up in the event of a crash, it is safe to do so. The requirement for immediate backup of database elements to disk can be avoided if we use a logging mechanism called redo logging. The principal differences between redo and undo logging are:

1. While undo logging cancels the effect of incomplete transactions and ignores committed ones during recovery, redo logging ignores incomplete transactions and repeats the changes made by committed transactions.
2. While undo logging requires us to write changed database elements to disk before the COMMIT log record reaches disk, redo logging requires that the COMMIT record appear on disk before any changed values reach disk.
3. While the old values of changed database elements are exactly what we need to recover when the undo rules U1 and U2 are followed, to recover using redo logging, we need the new values instead.

In redo logging the meaning of a log record <T, X , v> is "*transaction T wrote new value v for database element X*". There is no indication of the old value of X in this record. Every time a transaction T modifies a database element X , a record of the form < T ,X ,v> must be written to the log.

For redo logging, the order in which data and log entries reach disk can be described by a single "redo rule," called the **write-ahead logging rule**.

R1) Before modifying any database element X on disk, it is necessary that all log records pertaining to this modification of X , including both the update record < T ,X ,v> and the <COMMIT T> record, must appear on disk.

Example 10.6. Let's repeat the same example with this new logic.

Step	Action	t	MemA	MemB	DiskA	DiskB	MemLog	DiskLog
1							<Start T>	
2	$Read(A,t)$	8	8		8	8		
3	$t \leftarrow t \times 2$	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	$t \leftarrow t \times 2$	16	16	8	8	8		
7	$Write(B,t)$	16	16	16	8	8	<T,B,16>	
8							<Commit T>	
9								<Start T>;<T,A,16>;<T,B,16>;<C
10	$Output(A)$	16	16	16	16	8		
11	$Output(B)$	16	16	16	16	16		

The procedure to recover from a failure is different from that of undo logging. In undo logging, we discard uncommited changes, because we are unsure if they are done in the database. In redo logging, we proceed by doing again those changes that are commited, because these are now those that we are unsure about, while

uncommitted changes we know for sure that have not been made. The recovery rules for redo logging are illustrated in Algorithm 7.

Algorithm 7 Redo logging: recovery rules

```
Let S = set of transaction with <Ti,start> in log, but no
                <Ti,commit> not <Ti,end> in log

for each <Ti,X,v> in log in forward order do
        if Ti in S then
                write(X,v)
                output(X)

for each Ti in S do
        write <Ti,end> to log
```

Combining <Ti,end> records

There exist objects that are accessed often, which are usually called **hot objects**. One idea about this object is that as they are accessed very often, they would require lots of I/O operations and, many times, updated values of these objects would not even be needed to have been on disk, so we can try to delay writing them to disk as long as we can work with their values in memory. This way, we can just write their latest value and perform less I/O operations.

Example 10.7. Imagine we have four transactions:
T_1 : *..update X..*,T_2 : *..update X..*,T_3 : *..update X..*,T_4 : *..update X..* which can be executed with the following set of actions:

$$\begin{gathered} Write\,(X) \\ Output\,(X) \\ Write\,(X) \\ Output\,(X) \\ Write\,(X) \\ Output\,(X) \\ Write\,(X) \\ Output\,(X) \end{gathered}$$

And this way we would be updating X unnecessarily. A better way to handle this is:

$$\begin{gathered} Write\,(X) \\ \cancel{Output\,(X)} \\ Write\,(X) \\ \cancel{Output\,(X)} \\ Write\,(X) \\ \cancel{Output\,(X)} \\ Write\,(X) \\ Output\,(X) \\ combined\ < end > \end{gathered}$$

Nonetheless, there is an even better way to tackle this problem: **checkpointing**.

10.2.3 Checkpointing with undo logging

As we observed, recovery requires that the entire log be examined, in principle. When logging follows the undo style, once a transaction has its COMMIT log record written to disk, the log records of that transaction are no longer needed during recovery. We might imagine that we could delete the log prior to a COMMIT, but sometimes we cannot. The reason is that often many transactions execute at once. If we truncated the log after one transaction committed, log records pertaining to some other active transaction T might be lost and

could not be used to undo T if recovery were necessary. The simplest way to untangle potential problems is to checkpoint the log periodically. In a simple checkpoint, we:

1. Stop accepting new transactions.
2. Wait until all currently active transactions commit or abort and have written a COMMIT or ABORT record on the log.
3. Flush the log to disk.
4. Write a log record <CKPT>, and flush the log again.
5. Resume accepting transactions.

Example 10.8. An undo log with checkpointing:

$$
\begin{array}{c}
< Start\ T_1 > \\
< T_1, A, 5 > \\
< Start\ T_2 > \\
< T_2, B, 10 > \\
- - - - - - \\
< T_2, C, 15 > \\
< T_1, D, 20 > \\
< Commit\ T_1 > \\
< Commit\ T_2 > \\
< CKPT > \\
< Start\ T_3 > \\
< T_3, E, 25 > \\
...
\end{array}
$$

In the dotted line, a checkpoint was launched, so no more transactions are accepted to execute until T_1 and T_2 finish. When both transactions commit, we can now write the checkpoint and accept new transactions, such as T_3.

10.2.4 Checkpointing with redo logging

The steps to perform a checkpoint of a redo log are as follows:

1. Write a log record <START CKPT (Ti,... ,Tk)>, where Ti,... ,Tk are all the active (uncommitted) transactions, and flush the log.
2. Write to disk all database elements that were written to buffers but not yet to disk by transactions that had already committed when the START CKPT record was written to the log.
3. Write an <END CKPT> record to the log and flush the log.

Example 10.9. A redo log with checkpointing:

When we start the checkpoint, only T2 is active, but the value of A written by T1 may have reached disk. If not, then we must copy A to disk before the checkpoint can end.

$$
\begin{array}{c}
< Start\ T_1 > \\
< T_1, A, 5 > \\
< Start\ T_2 > \\
< Commit\ T_1 > \\
< T_2, B, 10 > \\
< Start\ CKPT\ (T_2) > \\
< T_2, C, 15 > \\
< Start\ T_3 > \\
< T_3, D, 20 > \\
< End\ CKPT > \\
< Commit\ T_2 > \\
< Commit\ T_3 >
\end{array}
$$

10.2.5 Undo/Redo logging

We have seen two different approaches to logging, differentiated by whether the log holds old values or new values when a database element is updated. Each has certain drawbacks:

- Undo logging requires that data be written to disk immediately after a transaction finishes, perhaps increasing the number of disk I/O operations that need to be performed.
- On the other hand, redo logging requires us to keep all modified blocks in buffers until the transaction commits and the log records have been flushed, perhaps increasing the average number of buffers required by transactions.
- Both undo and redo logs may put contradictory requirements on how buffers are handled during a checkpoint, unless the database elements are complete blocks or sets of blocks. For instance, if a buffer contains one database element A that was changed by a committed transaction and another database element B that was changed in the same buffer by a transaction that has not yet had its COMMIT record written to disk, then we are required to copy the buffer to disk because of A but also forbidden to do so, because rule R1 applies to B.

An **undo/redo log** has the same sorts of log records as the other kinds of log, with one exception. The update log record that we write when a database element changes value has four components. Record < T ,X ,v,w > means that transaction T changed the value of database element X; its former value was v, and its new value is w. The constraints that an undo/redo logging system must follow are summarized by the following rule:

UR1) Before modifying any database element X on disk because of changes made by some transaction T, it is necessary that the update record <T, X , v, w> appear on disk.

Example 10.10. An undo/redo log. Let's redo our typical example:

Step	Action	t	MemA	MemB	DiskA	DiskB	MemLog	DiskLog
1							<Start T>	
2	$Read(A,t)$	8	8		8	8		
3	$t \leftarrow t \times 2$	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	$t \leftarrow t \times 2$	16	16	8	8	8		
7	$Write\,(B,t)$	16	16	16	8	8	<T,B,8,16>	
8	$FlushLog$							<Start T>;<T,A,8,16>;<T,B,8,16>
9	$Output\,(A)$	16	16	16	16	8		
10							<Commit T>	
11	$Output\,(B)$	16	16	16	16	16		

Note that, in this case, the last three steps could've appeared in any order.

The undo/redo recovery policy is:

1. Redo all the committed transactions in the order earlierst-first.
2. Undo all the uncommitted transactions in the order latest-first.

11 Concurrency control

Interactions among concurrently executing transactions can cause the database state to become inconsistent, even when the transactions individually preserve correctness of the state, and there is no system failure. Thus, the timing of individual steps of different transactions needs to be regulated in some manner. This regulation is the job of the **scheduler** component of the DBMS, and the general process of assuring that transactions preserve consistency when executing simultaneously is called **concurrency control**.

11.1 Schedules: serial, serializable and conflict-serializable

Definition 11.1. A **schedule** is a sequence of the important actions taken by one or more transactions.

Example 11.1. Imagine we have the constraint $A = B$ and the following two transactions:

$$
\begin{array}{ll}
T_1: Read(A) & T_2: Read(A) \\
\quad A \leftarrow A + 100 & \quad A \leftarrow A \times 2 \\
\quad Write(A) & \quad Write(A) \\
\quad Read(B) & \quad Read(B) \\
\quad B \leftarrow B + 100 & \quad B \leftarrow B \times 2 \\
\quad Write(B) & \quad Write(B)
\end{array}
$$

Note how each of the transactions individually preserves the consistency of the database. Nonetheless, there exist some schedules that can make things go wrong!

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(B); A<-A*2		
	Write(A);	250	
	Read(B); B<-B*2		
	Write(B);		250
		250	250

This schedule poses no problems.

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

This schedule poses no problems. Note, nonetheless, how the different orderings affect the final result.

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

This schedule poses no problems.

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

But this schedule is problematic! The final state of the database is inconsistent.

We want schedules that are '*good*', in the sense that they ensure the final state of the database to be consistent, independent of the transactions' semantics and the initial (consistent) state of the database.

For this, we should only look at the order of read and writes.

A schedule can be represented as its sequence of actions, where $r_i(X)$ means X is read in transaction i and $w_i(X)$ means X is written in transaction i.

For example, schedule C can be represented as

$$S_C = r_1(A)\, w_1(A)\, r_2(A)\, w_2(A)\, r_1(B)\, w_1(B)\, r_2(B)\, w_2(B).$$

Definition 11.2. A schedule is **serial** if its actions consist of all the actions of one transaction, then all actions of another transaction, and so on.

For example, Schedules A and B are serial.

Remark 11.1. Note that all serial schedules work: they leave a consistent database, because of the correctness principle.

Definition 11.3. A schedule S is serializable if there is a serial schedule S' such that for every initial database state, the effects of S and S' are the same.

For example, Schedule C is serializable, with $C' = A$, but schedule D is not serializable, because it leads to an inconsistent state.

The transaction game

It is a way to visually check for serializability (only for simple schedules).

Example 11.2. For example, Schedule C is:

A	r	w	r	w				
B					r	w	r	w
T1	r	w			r	w		
T2			r	w			r	w

Here, we represents taken by each transaction and the affected variable, in the order of the schedule. Now, two steps can be exchanged if they are next to one another and they can slide without colliding.

A	r	w	r	w				
B					r	w	r	w
T1	r	w			r	w		
T2			r	w			r	w

Here, the red colored steps cannot be exchanged, because they collide in the first row.

A	r	w	r	w				
B					r	w	r	w
T1	r	w			r	w		
T2			r	w			r	w

But now, the green colored steps can be exchanged, because they are next to each other and they do not collide if we slide the columns:

A	r	w			r	w		
B			r	w			r	w
T1	r	w	r	w				
T2					r	w	r	w

And this is schedule A!

Example 11.3. Now, for schedule D:

A	r	w	r	w				
B					r	w	r	w
T1	r	w					r	w
T2			r	w	r	w		

Everything is red!

Let's continue the discussion to try to understand why D is different from C.

Definition 11.4. We define **conflicting actions** as those that cannot be reordered in a schedule. These are:

- Obviously, actions made by the same transactions cannot be reordered.
- But there are also actions from different transactions whose reordering would affect the result:
  - $r_1(A)$ and $w_2(A)$ cannot be reordered
  - $w_2(A)$ and $r_1(A)$ cannot be reordered
  - $w_1(A)$ and $w_2(A)$ cannot be reordered

Definition 11.5. A schedule is **conflict serializable** if it can be serialized without violating any conflicting action. If S is conflict serializable to S', then S and S' are **conflict equivalent**.

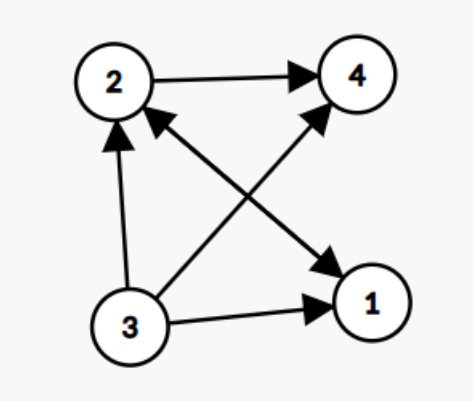
These definitions allow us to create a mathematical tool for determining '*good*' schedules, understanding that as a conflict serializable schedule. This goal is achieved by a precedence graph:

Definition 11.6. A **precedence graph** of a schedule S, $P(S)$, is graph $P(S) = (N, E)$ where:

- The nodes, N, are the transactions in S.
- The arcs are directed, and $(T_i, T_j) \in E$ whenever:
  - $p_i(A), q_j(A)$ are actions in S
  - $p_i(A) <_S q_j(A)$, i.e., $p_i(A)$ precedes $q_j(A)$
  - at least one of p_i, q_j is a write

Example 11.4. Let's compute $P(S)$ for

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



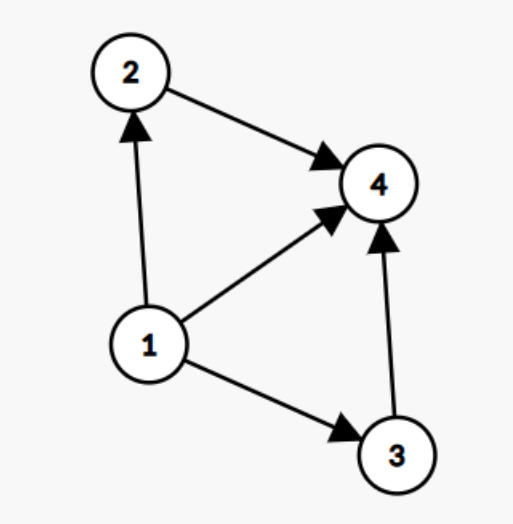
How to use precedence graphs to determine conflict-serializability?

To tell whether a schedule S is conflict-serializable, construct the precedence graph for S and ask if there are any cycles. If so, then S is not conflict-serializable. But if the graph is acyclic, then S is conflict-serializable.

Example 11.5. The schedule in Example 11.4 is not conflict serializable, because there is cycle between nodes 1 and 2.

Example 11.6. Is $S = w_1(A)\, r_2(A)\, r_3(A)\, w_4(A)$ conflict serializable?

First, construct the precedence graph:



In this case, there are no cycles, so S is conflict serializable.

11.2 How to enforce serializability: locking

11.2.1 Option 1: let luck be our friend

Run the system, recording the precedence graphs of the schedules used. At the end of the day (at some decided point in time), check for $P(S)$ cycles and declared if the execution was good.

Of course, this is not a very good option, because we are letting luck decide whether we are losing time and energy or not.

11.2.2 Option 2: a locking protocol

Prevent cycles from occurring!

Now, we define two new actions:

- LOCK(A), $l_i(A)$: lock access to object A, i.e., only transaction i can access object A both for read and write.

- UNLOCK(A), $u_i\,(A)$: unlock A so it can be accessed by other transactions.

Definition 11.7. Rule 1: A transaction is **well-formed** if before performing any operation on an object A, it has locked it before, and it unlocks it afterwards.

Definition 11.8. Rule 2: A schedule is **legal** if all objects that are locked, have been previously unlocked (or never locked before).

Example 11.7. Let's analyze some schedules in terms of well-form and legality:

- S1=l1(A)l1(B)r1(A)w1(B)l2(B)u1(A)u1(B)r2(B)w2(B)u2(B)l3(B)r3(B)u3(B) ç

  Let's start assessing if all transactions are well-formed:

  S1=l1(A)l1(B)r1(A)w1(B)l2(B)u1(A)u1(B)r2(B)w2(B)u2(B)l3(B)r3(B)u3(B)

  As we can see, all actions of each transaction are preceded by a lock to the pertinent object and an unlock afterwards, so all transactions are well formed.

  Regarding legality:

  S1=l1(A)l1(B)r1(A)w1(B)l2(B)u1(A)u1(B)r2(B)w2(B)u2(B)l3(B)r3(B)u3(B)

  We can see how object B is locked by transaction 2 before being unlocked by transaction 1, so the schedule is not legal.

- S2=l1(A)r1(A)w1(B)u1(A)u1(B)l2(B)r2(B)w2(B)l3(B)r3(B)u3(B)

  Regarding well-form:

  S2=¿?l1(A)r1(A)w1(B)u1(A)u1(B)l2(B)r2(B)w2(B)l3(B)r3(B)u3(B)¿?

  B is modified by transaction 1 before being locked, so transaction 1 is not well formed.

  B is locked, read and written by transaction 2, but it is never unlocked, so transaction 2 is also not well formed.

  Transaction 3 is well formed.

  Regarding legality, we can see how B is locked by transaction 3 before being unlocked by transacion 2, so the schedule is not legal.

- S3=l1(A)r1(A)u1(A)l1(B)w1(B)u1(B)l2(B)r2(B)w2(B)u2(B)l3(B)r3(B)u3(B)

  This schedule is obviously legal and all transactions are well formed.

Definition 11.9. Rule 3: The **two phase locking (2PL)** scheme refers to a strategy for scheduling in which there are no unlocks for transaction T_i until all locks for T_i have been acquired, and there are no locks for T_i after any unlock for T_i, i.e., all locks are acquired before any unlock is performed.

Theorem 11.1. *All schedules verifying rules 1,2 and 3 are conflict-serializable. Meaning, if a schedule has all its transactions well-formed, the schedule is legal and it uses the 2PL locking scheme, then the schedule is conflict-serializable.*

Remark 11.2. Note that the converse is not true: there are conflict-serializable schedules that are not 2PL.

Beyond the 2PL protocol, it is all a matter of improving performance and allowing more concurrency, for these, there exist more artifacts, such as shared locks, multiple granularity locks,...

11.3 Shared locks

Until now, we are locking an object before any action can be applied to it, but sometimes it is possible to actually grant access to several transactions to interact with the same object, if the actions that they need to perform are not in conflict. For example, if two transactions want to read from the same variable, they can do it without problem, but with our scheme we don't allow this.

A way to amend this is to define **shared locks** which are locks that can be shared by several transactions, provided they only want to read the object.

We define new actions:

- EXCLUSIVE-LOCK(A), $xl_i(A)$: lock object A in exclusive mode, i.e., no other transaction can lock it until it is unlocked.
- SHARED-LOCK(A), $sl_i(A)$: lock object A in shared mode, i.e., other transactions can lock it in shared mode, but not in exclusive mode.
- EXCLUSIVE-UNLOCK(A), $xu_i(A)$: unlock A from exclusive lock.
- SHARED-UNLOCK(A), $su_i(A)$: unlock A from shared lock.

Note that it is usual to just represent both unlocks by $u_i(A)$, assuming the computer will execute the correct action depending on the state of the object.

Now, we have to redefine our rules:

Definition 11.10. Rule 1: a transaction is **well-formed** if:

- Before performing a read action on an object, it has been previously locked, exclusively or shared (but better if it is shared locked, allowing for increased performance).
- Before performing a write action on an object, it has been previously exclusively locked.

Rule 2: a schedule is **legal** if:

- No exclusive lock is performed on a locked (exclusive or shared) object, until it has been unlocked.
- No lock (exclusive or shared) is performed on a exclusively locked object, until it has been unlocked.

This rule can be summarize in a **compatibility matrix**, which shows which state changes are allowed:

	T_j asks for		
		S	X
T_i holds	S	True	False
	X	False	False

Rule 3: a schedule respect the **2PL protocol** if for any two-phase locked transaction T_i, no action $xl_i(A)$ or $sl_i(A)$ is preceded by an action $u_i(B)$ for any object B.

Notice that there are transactions that read and write the same object, so what should we do about this? There are two main options:

1. We can just request an exclusive lock from the beginning.
2. We can use an **upgrade** scheme, in which a shared lock is acquired if we are unsure if we will need to write the object later. When we need to write it, we '*upgrade*' the shared lock to be an exclusive lock. This can be technically achieved by allowing transactions to have two locks on the same object: one shared and one exclusive; or by releasing the shared lock and getting the exclusive lock[7].

[7] Note that in this case we have to slightly modify Rule 3 to let transactions release shared locks to get exclusive locks in the locking phase.

11.4 More types of locks

11.4.1 Increment lock

We can define a new action, which is an **atomic increment action**, as

$$IN_i(A,k) \equiv \{Read(A); A \leftarrow A + k; Write(A)\}.$$

A property of these actions is that they are commutative, so they do not conflict between each other, even though they involve writing the objects. This allows for more flexibility, because we can define a new lock, the **increment lock**:

- INCREMENT-LOCK(A), $il_i(A)$: lock object A to perform increment actions on it.

And we have to, again, redefine the rules:

Definition 11.11. Rule 1: a transaction is **well-formed** if:

- Before performing a read action on an object, it has been previously locked, exclusively or shared (but better if it is shared locked, allowing for increased performance).
- Before performing a write action on an object, it has been previously exclusively locked.
- Before performing an increment action on an object, it has been previously increment-locked.

Rule 2: a schedule is **legal** if:

- No exclusive lock is performed on a locked (exclusive, increase or shared) object, until it has been unlocked.
- No lock (exclusive, increase or shared) is performed on a exclusively locked object, until it has been unlocked.
- An increase lock can only be performed on unlocked objects or increase-locked objects.

This rule can be summarize in a **compatibility matrix**, which shows which state changes are allowed:

	T_j asks for			
T_i holds		S	X	I
	S	True	False	False
	X	False	False	False
	I	False	False	True

Rule 3: a schedule respect the **2PL protocol** if for any two-phase locked transaction T_i, no action $xl_i(A)$, $sl_i(A)$ or $il_i(A)$ is preceded by an action $u_i(B)$ for any object B.

11.4.2 Update lock

A common deadlock problem that arises when we use upgrading shared locks is depicted below:

T1	T2
$sl_1(A)$	
	$sl_2(A)$
$lx_1(A)$	
	$lx_2(A)$

In this case, both transactions are waiting for the other to release the object A to be able to lock it exclusively, so there is a deadlock.

The **solution** implies decreasing the level of concurrency, but it is worthy to avoid such problematic cases. We define a new lock, in which a transaction which is unsure about if it will need to write an object, it acquire an **update lock** instead of the shared lock, and upgrades can only be made from this lock:

- UPDATE-LOCK(A), $ul_i(A)$: lock object A and might upgrade later.

Note that if an object is shared-locked, it can be update-locked, but not the other way round (if we allow this, the behavior would not change).

Let's redefine our three rules for this case:

Definition 11.12. Rule 1: a transaction is **well-formed** if:

- Before performing a read action on an object, it has been previously locked, exclusively, update or shared (but better if it is shared locked, allowing for increased performance).
- Before performing a write action on an object, it has been previously exclusively locked.
- Before upgrading a lock, it has been previously update-locked.

Rule 2: a schedule is **legal** if:

- No exclusive lock is performed on a locked (exclusive, update or shared) object, until it has been unlocked.
- No lock (exclusive, update or shared) is performed on a exclusively locked object, until it has been unlocked.
- An update lock can only be performed on unlocked objects or shared-locked objects.

This rule can be summarize in a **compatibility matrix**, which shows which state changes are allowed:

	T_j asks for			
T_i holds		S	X	U
	S	True	False	True
	X	False	False	False
	U	False	False	False

Note that in this case, an object can be locked in several modes (for instance, it can be locked in shared and update mode at the same time), so transitions are based on the most restrictive lock mode.

Rule 3: a schedule respect the **2PL protocol** if for any two-phase locked transaction T_i, no action $xl_i(A)$ or $sl_i(A)$ is preceded by an action $u_i(B)$ for any object B.

11.5 Lock granularity

We have been talking about locking objects, but we have not specified which are these objects. They can be tuples, pages, relations,... In all cases the scheme works, but choosing the appropriate size for what we are locking is obviously going to affect performance. For instance:

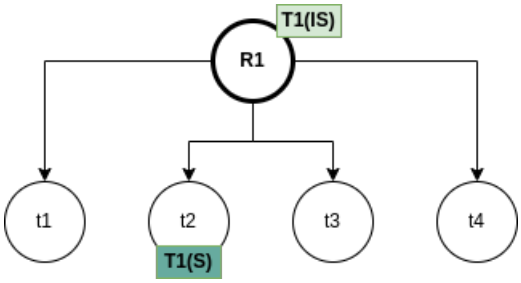
- If we lock at tuple level, we gain in concurrency capabilities, but we will need to increase the efforts to control the concurrent access. For example, the memory needed to store all locks would increase.
- If we lock at relation level, it is easier to address the concurrency issues, but we lose many concurrency capabilities because as soon as some transaction is trying to modify one tuple of one relation, the whole relation would be inaccessible for other transactions.

We can define **multi-granular locks**, which can specify at what level they want to lock the objects involved.

For this, we define **intentional locks**, which can be of any of the types we have seen, but with a slightly different meaning:

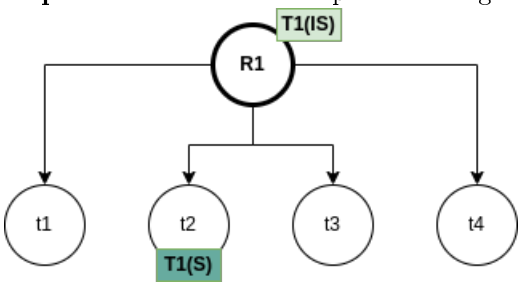
- An intentional lock on an object, A, is trying to obtain a lock on a subobject, A_s.
- Intentional locks indicate the type of the lock of the subobjects.

Example 11.8. Imagine we have the relation R_1 which has four tuples. If transaction 1, T_1, wants to shared-lock the second tuple, t_2, we need to obtain an intentional shared-lock on R_1 and then a shared-lock on t_2. This is depicted below:

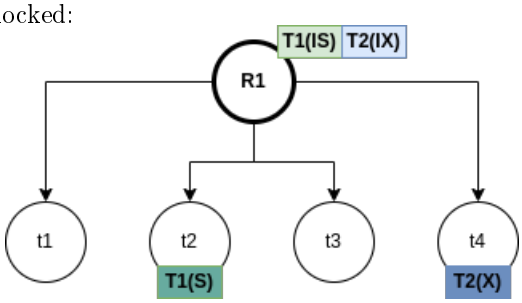


Now, imagine T_2 wants a shared-lock in the whole relation, then, it will can be acquired, because the compatibility matrix allows it. But it wanted an exclusive-lock in the whole relation, it would need to wait until the locked tuple is unlocked.

Example 11.9. Another example is starting with the previous one:



Now, imagine T_2 wants to exclusively lock t_4: this can be done, because t_4 is unlocked. In this case, T_2 needs to ask for an intentional exclusive lock on R_1, which would be granted because the relation R_1 is not fully locked. Then at tuple level, T_2 would ask for an exclusive lock for t_4, an it would be granted because it is unlocked:



Again, we can build the compatibility matrix for this new kind of locks:

		T_j asks for				
		IS	IX	S	SIX	X
T_i holds	IS	True	True	True	True	False
	IX	True	True	False	False	False
	S	True	False	True	False	False
	SIX	True	False	False	False	False
	X	False	False	False	False	False

- IS, intentional shared lock: lock some subobjects in share mode.
- IX, intentional exclusive lock: lock some subobjects in exclusive mode.
- S, shared lock: lock an object in shared mode.
- SIX, shared intentional exclusive lock: lock an object in shared mode and some subobjects in exclusive mode.
- X, exclusive lock: lock an object in exclusive mode.

Also, there are restrictions in which states can subobjects have in terms of the states of the parent object:

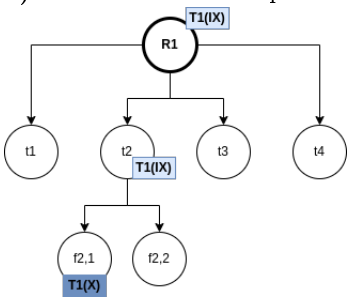
Parent state	Child possible states
IS	IS,S
IX	IS,S,IX,X,SIX
S	none
SIX	X,IX
X	none

The rules to follow are the following:

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

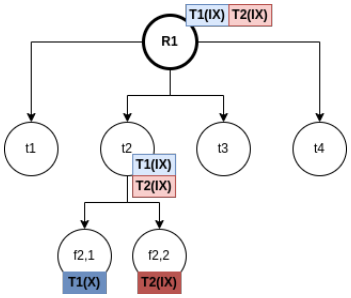
Example 11.10. Let's do some practice.

1) Start with this setup:

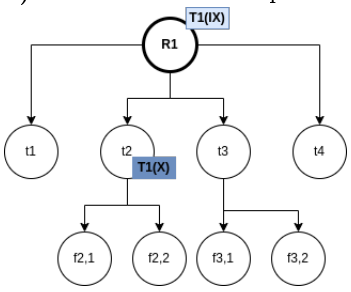


Can T_2 access object f_{22} in mode X? If so, what locks would T_2 get?

Yes, it can, because all the locks in the sequence are compatible with another IX lock and $f_{2,2}$ is unlocked:



2) Start with this setup:

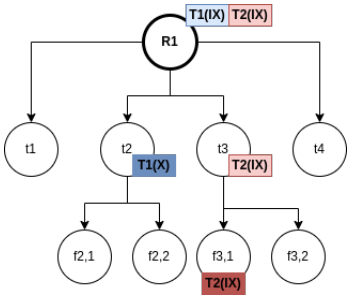


Can T_2 access object f_{22} in mode X? If so, what locks would T_2 get?

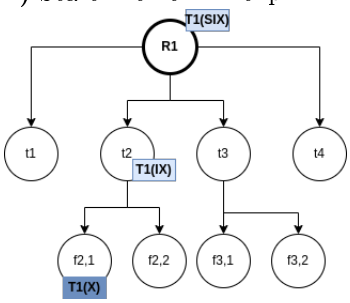
No, it cannot, because parent t_2 is in X mode, so it cannot be locked in IX mode by T_2.

3) From the last setup: can T_2 access object $f_{3,1}$ in X mode? What locks would T_2 get?

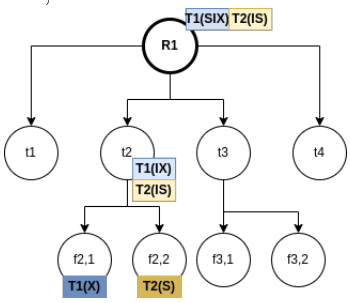
Yes, it can, because $f_{3,1}$ is unlocked, its parent t_3 is unlocked, and its parent is in IX state, compatible with another IX state:



4) Start with this setup:



Can T_2 access object $f_{2,2}$ in S mode? What locks would T_2 get?
Yes, because SIX and IX and compatible with IS:



5) In the previous setup: can T_2 access object $f_{2,2}$ in X mode? What locks would T_2 get?
No, because SIX is not compatible with IX.

Part VII
Distributed Databases

12 Distributed databases

Distributed database management systems distribute and replicate data over multiple machines to try to meet the availability, durability, performance, regulatory amd scale requirements of large organizations, subject to physics.

A distributed database does two things:

- **Distribution**: place partitions of data on different machines.
- **Replication**: place copies of data on different machines.

The **goal** is to offer the same functionality and transactional semantics as a RDBMS with distributed features.

The **reality** is that there need to be done concessions in terms of functionality, transactional semantics and performance.

There are three main challenges in distributed databases, which are how to distribute the data, how to access the data and how to perform distributed transactions.

Definition 12.1. A **shard** is a horizontal partition of data in a database.

12.1 Data distribution

There are several ways to distribute the data:

- **Range distribution**: tables are partitioned by a *distribution key*, which is usualy part of a primary key. Each shard contains a range of the values.

  Example 12.1. Imagine a CUSTOMER table that is distributed in two servers: in one server we maintain all names starting by a letter in the range A-N and in the other server all starting with a letter in the range M-Z.

- **Hash distribution**: there two ways to implement this:
  - Each shard contains a modulo of a hash value.
  - Each shard contains a range of hash values.

In the context of data distribution, it is usually desirable to maintain approximately the same amount of data in each node. To achieve this, **rebalancing** is used. Rebalancing encompasses:

- Moving shards to achieve better data distribution across nodes.
- Splitting shards to achieve better data distributions across nodes.

Another important concept is that of **co-location**, which makes use of the fact that some tables share some attributes. If a shared attribute is related to the distribution key, then we can store different tables in a distributed manner, in such a way that if we perform a join between these tables using this attribute, we minimize the interaction between tables from different nodes.

Other times, for the same purpose of increasing the efficiency of distributed joins it is useful to replicate small tables to enable fast joins, foreign keys and other operations. This technique is called **reference tables**.

There are more ways to tackle data distribution:

- Use random distributions to distribute the data.
- Use list distribution, assigning labels to the different partitions.
- Use spatial distribution, locating data in the servers where it will be of use.

12.2 Distributed data access: distributed SQL

To scale query throughput linearly with the number of nodes, queries should only access one node. The techniques of co-location and reference table enable relatively complex queries. This idea of using the nodes information inside the queries to only access the desired nodes is called **routing queries**.

Example 12.2. A routing query trying to access only the node where the distribution key is 36:

```
INSERT INTO dist1(dist_key, value) VALUES(36, 11);

SELECT *
FROM dist1
WHERE dist_key=36 AND value<11;

UPDATE dist1
SET value=3
WHERE dist_key=36 AND value<11;
```

Nonetheless, the relational algebra can be extended to work in distributed system: it is called the **multi-relational algebra**, and it adds to the usual relational algebra the operations[8]:

- COLLECT: takes data from several nodes and single output data stream combining all of them.
- REPARTITION: takes data from some nodes and sends it to other nodes.
- BROADCAST: sends its input rows to multiple consumers on demand. Each consumer gets all of the rows.

As in standard SQL, in distributed SQL it is needed to perform logical planning of the queries before executing them, with the need that the final result is the same as it would be if all data were in one node.

Example 12.3. Imagine we want to compute the mean of an attribute in a table which is distributed in different nodes. The steps to follow would be:

1. In each node: SUM(A) and COUNT(A)
2. In requesting node:
   (a) COLLECT all data: sum_ = SUM(SUM(N->T.A), N in Nodes), count_ = SUM(COUNT(N->T.A), N in Nodes)
   (b) mean = sum_ / count_

Also, different plans can be defined and the best one should be chosen following some optimization criteria.

Co-located joins

Imagine we have the query:

```
SELECT dist1.dist_key, count(*)
FROM dist1
JOIN dist2 ON (dist1.dist_key = dist1.dist_key)
WHERE dist2.value < 44
GROUP BY dist1. dist_key;
```

One way to do this is:

1. In each node:
   (a) Scan dist1
   (b) Scan dist2 and filter dist2.value < 44

[8] The definitions have been taken from MSQLS Docs, but the names are different.

2. In requesting node:
   (a) Collect all dist1
   (b) Collect all filtered dist2
   (c) Perform the join
   (d) Aggregate

But as we are using the distribution key for joining, we know that the tables that will join are stored together, so we can make use of the co-location by changing the plan to:

1. In each node:
   (a) Scan dist1
   (b) Scan dist2 and filter dist2.value < 44
   (c) Perform the join
   (d) Aggregate
2. In requesting node:
   (a) Collect the values

For this changes to work, we are using that: filter is commutative with collect, group by dist_key is commutative with collect and join is co-located, so it is commutative with collect.

When we do this, we are working with much smaller tables, thus reducing the response time.

Re-partition joins

Now imagine the query:

```
SELECT dist1.dist_key, count(*)
FROM dist1
JOIN dist2 ON (dist1.dist_key = dist2.other_key
WHERE dist2.value < 44
GROUP BY dist1.dist_key;
```

In this case, the initial plan also work, but again we can do better repartitioning the dist2 tables to the nodes where they will be needed. This might seem like too heavy work to do, but if the tables are large enough, the whole relations might not even fit in only one node, so this can be the only way to be able to produce the response. The alternative plan is:

1. In each node:
   (a) Scan dist2
   (b) Filter dist2.value < 44
2. In each node:
   (a) Repartition filtered tuples from dist2 with corresponding other_key to dist_key of this node
   (b) Scan dist1
   (c) Perform the join
   (d) Aggregate
3. In requesting node:
   (a) Collect the values

Broadcast joins

Now, imagine the query:

```
WITH top10 AS (
        SELECT dist_key, count(*)
        FROM dist1
        GROUP BY 1
        ORDER BY 2
        LIMIT 10
        )

SELECT *
FROM dist2
WHERE other_key IN (SELECT dist_key FROM top10);
```

The naïve plan is:

1. In each node:
   (a) Scan dist1
   (b) Scan dist2
2. In requesting node
   (a) Collect dist1
   (b) Aggregate dist1
   (c) Sort/limit dist1
   (d) Collect dist2
   (e) Perform the join

But this can be improved by creating a subplan that handles order/limit under join and broadcasting the subplan to pull the collect above the join. The idea is that TOP10 among all the data is the same that the TOP10 among all the TOP10s in each node:

1. In each node:
   (a) Scan dist1
   (b) Preaggregate: get TOP10 of the node
2. In requesting node:
   (a) Collect TOP10 of each node
   (b) Merge all TOP10
   (c) Sort/limit and get final TOP10
   (d) Broadcast this TOP10 to all nodes
3. In each node:
   (a) Scan dist2
   (b) Join dist2 with broadcasted TOP10
4. In requesting node:
   (a) Collect the values

Observations

As we have seen, the query plans depend heavily on the distribution key.

Runtime also depends on the query, data, the data size, network speed,...

This menas distributed databases require adjusting the distribution keys and queries to each other to achieve high performance.

12.3 Distributed transactions

Ideally, we have ACID transactions: Atomicity, Consistency, Isolation and Durability.

The main distribution challenges are:

- Atomicity: commit on all nodes or none of them.
- Isolation: see other distributed transactions as commited/aborted

Additionally, it is important to have a mechanism for distributed deadlock detection.

12.3.1 Atomicity

Atomicity is achieved through 2PC (2-Phase Commit):

Phase 1: Store transactions on all nodes

Phase 2: Store final commit decision and

(a) If success: commit all stored transactions

(b) If error: abort all prepared transactions

(Recovery phase): Commit/abort prepared transactions after system failure

12.3.2 Isolation

If we query different nodes at different times, we may see a concurrent transaction as commited on one node, but not yet committed on another one.

Distributed snapshot isolation means we have the same view of what is commited and what not on all nodes.

We must also ensure **consistency**: any preceding write is seen as committed.

A common approach is to add a timestamp to each query, so queries see all commits with lower timestamps.

There are different ways of dealing with clock synchronization:

- **TrueTime**: synchronize clocks using GPS/atomic clocks. Commits pause until all clock move past commit time.
- **Clock-SI**: queries collect current time from all nodes involved, pick the highest timestamp and wait for it to pass.
- **HLC (Hybrid Logical Clocks)**: they are increased whenever an event occurs or a message from another node is received with a higher timestamp.

12.3.3 Considerations

Fully resolving a 2PC might take time in case of system failure.

Distributed deadlock detection is essential to stability, but not always implemented.

Snapshot isolation avoids seeing partially committed transactions, but at a cost, and read-your-writes consistency can be at risk.

12.4 Replication

Replication means to store the same data in different nodes. It can be useful for:

- Availability: resume from replica in case of node failure.
- Durability: restore from replica in case of disk failure.
- Read throughput: divide reads across read replicas.
- Read latency: local/nearby replica gives lower read latency.
- Write latency: local/nearby replica gives lower write latency.

12.4.1 Quorums

The basic idea is to read from R nodes and write to W nodes, where R+W>N, being N the total number of nodes.

The challenge is to apply events in the same order wverywhere.

12.4.2 Follow the leader

We can also assign a temporary leader to serialize writes efficiently. This leader would then feed the rest of the nodes with the new data.

If one node fails (**standby fail**), the leader will continue writing to other replica.

If the leader fails (**primary fail**), a failover is initiated, a replica is promoted to leader and the rest of the replicas follow the new leader.

12.4.3 N-directional

All nodes accept writes and then decide how to reconcile conflicting changes.

12.5 CAP theorem

Basically, the CAP theorem states that a database can only have two out of the three properties: **C**onsistency, **A**vailability and **P**artitioning. In the case of distributed databases, partitioning is a must, so one must decide between C and A (note, nonetheless, that even though the theorem ensure that one cannot get the three properties in their most strict form, it is possible to have them in relaxed, yet good, ways). So, basically, according to this theorem, we need to decide between:

- Availability (AP): keep writing to a minority of nodes, and the majority does not see it.
- Consistency (CP): make writes/reads temporary unavailable because consistency must be preserved.

But this is an incomplete picture of the trade-offs that appear in a distributed database.

12.6 PACELC theorem

This is an improved version of the CAP theorem, but is still oversimplified:

- If network **P**artition: choose **A**vailability or **C**onsistency.
- **E**lse: choose **L**atency or **C**onsistency.

12.7 More trade-offs

- Consistency: read-your-writes, no lost updates, linearizability
- Availability: for reads, for writes, handle availability zone failure
- Partition-tolerance: for reads, for writes.
- Durability: node failure does not result in data loss, writes are archived in a timely manner.
- Low latency: low read latency, low write latency, global latency VS local latency.
- Complexity: dependencies on other systems, multiple node types, optimizations.

References

[1] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System r: Relational approach to database management. *ACM Transactional Database Systems.*

[2] Hector Garcia-Molina, Jeffrey Ullman D, and Jennifer Widom. *Database Systems: The Complete Book.* 01 2002.

[3] Mahmoud Sakr. Infoh417 database systems architecture. Lecture Notes.

[4] Jan Van den Bussche and Stijn Vansummeren. Translating sql into the relational algebra.