CHAPTER 1

RELATIONAL DESIGN

1.1 Database Fundamentals

Human activity is data-driven. We are making decisions, proceeding with actions and reasoning, based on observed data. However, we are limited in storing all data in memory and recalling them. So, we define a robust base that can store, update and delete and search data efficiently. This is a database.

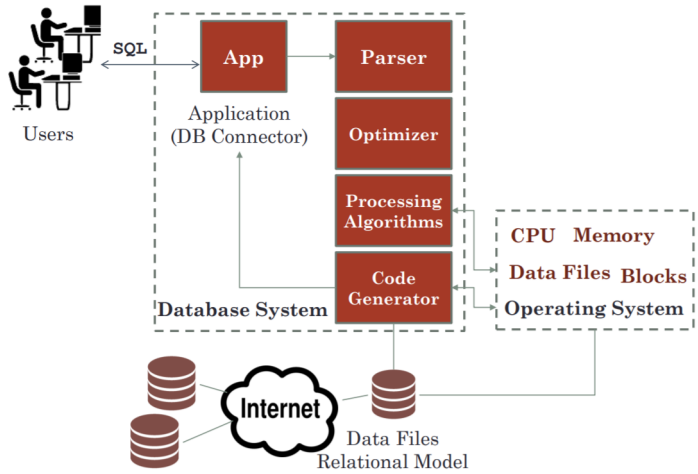
Data Management System

The fundamental functionality of a data management system is to:

- provide a data model, e.g. relational data model, object-oriented data model, etc.
- provide access to data, i.e. query, insert, delete and update
- provide tools to analyse data, i.e. complex aggregation queries (e.g. count the number of queries that satisfy a condition), histograms, etc.
- store data physically, from memory to hard disks.
- store data securely, by using control access to sensitive and confidential data and by encoding data.
- maintain data consistency, e.g. in the face of failures (e.g. software bugs, power cuts), and recover from failures.
- optimise data access to efficiently retrieve data, using indexing and hashing data structures along with optimisation algorithms.

In terms of the user, the database system is a black box with an interface for users/applications offering the functionalities we discussed. We allow the user to write commands in high-level language that can be translated into machine code. To do this, we make use of a declarative programming language (e.g. SQL). This allows us to manage and query data. It is declarative since we tell the database what to do, and not how to do it, e.g. we say find a tuple that matches the query, not what kind of searching algorithm to use in order to find the tuple.

In terms of systems engineering, the database system is a set of interconnected components, as shown below.



When a user queries the database using SQL, we parse and optimise the query, and then processes the algorithm (which talks to the CPU, memory, data files). Then, the operating system generates the code that we can run to the data files in order to retrieve the relevant tuples. We might require the internet to access the data files.

Data

There are 3 families of data:

- structured data- well-defined data structure, e.g. tables where we understand what the data represents;
- unstructured data- less information is provided on interpreting such data, e.g. web pages, texts, sensor measurements.
- semi-structured data- self-descriptive data, e.g. XML or JSON.

Modern data management systems can manage all families of data.

1.2 Relational Model

We will transform a textual description of a real problem into a set of concepts conveying exactly the same information. There are 2 ways of doing this: Entity-relationship (ER) modelling, and Relational modelling.

ER modelling does not guarantee optimality in operations and query executions. On the other hand, relational modelling is mathematics-driven. It uses relational algebra, set theory and functional dependency theory. This guarantees query optimisation based on the 4 fundamental operations (search, update, add, delete).

A conceptual data model is a mathematical model for interpreting our data. The interpretation involves the following concepts:

- the entities, e.g. students, employees;
- the attributes, e.g. name, address;
- the relationships, e.g. an employee works in a department, a student attends many courses.

In the relational conceptual model, we allow any entity to relate to other entities given that they share a common attribute(s) (i.e. the foreign key). For example, we can model books being borrowed in the library with 2 entities- readers and books. A reader has attributes: name, age, book title and id. A book has attributes: book title, subject, ISBN, number of pages. We can create a relation between books and readers using the common attribute book title.

Any entity and relationship are both called relations. It can be thought of as a 2-dimensional table. A column in this table is an ordered collection of attributes, while a row in this table is a tuple that represents an instance of the relation. Moreover, there has to be an attribute that uniquely identifies each tuple.

By representing data as tables, we can easily perform operations on it. In particular, we can list attributes of interest to be retrieved, and/or constrained attributes to filter out irrelevant tuples. For example, a query can to be return the names of the customers with active accounts and balance greater than £500. So, the attribute of interest is name, and the constrains are: active accounts and balance greater than £500. We can represent this in SQL as well.

```
SELECT Name
FROM BankAccount
WHERE Balance > 500 AND Status = "active"
```

Clearly, the language is declarative- we merely specify what to find, not how to find it.

Now, we will formalise the relational model. The schema of a relation is $R(A_1, A_2, \ldots, A_n)$. Here, R is the name of the relation, and the values $A_1, A_2, \ldots, A_n$ are a tuple of attributes. Each attribute A_i assumes values in a domain D_i. For example, a schema is:

`BankAccount(AccountNumber, Name, Balance, Status),`

where:

- the value of `accountNumber` is a natural number;

- the `name` is a character with length at most 50;
- the value of the `balance` is a real number; and
- the `status` is either `active` or `inactive`- it is an enumerated type.

A tuple of a relation R is an order set of values corresponding to the attributes of R satisfying the domain constrains (i.e. the value of tuple at index i lives in the domain D_i). It is denoted $t = (v_1, v_2, \ldots, v_n)$, with $v_i \in D_i$. We can also directly access the value of the attribute v_i by $t[v_i]$. An instance $r(R)$ is a set of tuples.

The relational model also allows the value of an entity to be `null`. We use this value to represent a value that is either unknown, inapplicable, uncertain, or missing. So, the relational database scheme is a set of relations $S = \{R_1, R_2, \ldots, R_n\} \cup \{\texttt{NULL}\}$.

Relational constraints

Relational constraints are conditions that must hold on all instances for each relation. There are 3 fundamental constraints:

- Key constraint- a key uniquely identifies a tuple in a relation;
- Entity integrity constraint- keys cannot be null;
- Referential integrity constraint- for two relations to have a relationship, they must share attribute(s).

A superkey (SK) of a relation R is a set of attributes that contains at least one attribute that uniquely identifies a tuple. That is, for $t_1, t_2 \in r(R)$,

$$t_1 \neq t_2 \implies t_1[SK] \neq t_2[SK].$$

For example, consider the following relation

`Employee (SSN, EName, LName, BDate, Salary, DNO).`

Then, the subsets:

- $\{\texttt{SSN}, \texttt{EName}, \texttt{BDate}\}$ contains `SSN`, which is unique for each attribute- it is a superkey;
- $\{\texttt{SSN}\}$, is also a superkey;
- $\{\texttt{EName}, \texttt{Salary}\}$ might not identify a tuple uniquely- it is not a superkey.

A candidate key is the minimal superkey, i.e. the set with the smallest number of attributes that can uniquely identify tuples. Formally, the set $K = \{A_1, A_2, \ldots, A_k\}$ is a candidate key if K is superkey and for all $j \in \{1, 2, \ldots, k\}$, $K \setminus \{A_j\}$ is not a superkey. That is, removing any attribute from K will mean that it is no longer a superkey.

In the example above, $\{\texttt{SSN}, \texttt{EName}, \texttt{BDate}\}$ is not a candidate key since $\{\texttt{SSN}, \texttt{BDate}\}$ is still a superkey. In fact, the only candidate key is $\{\texttt{SSN}\}$. If a relation has several candidate keys, then we can arbitrarily chosen one of them

to be a primary key (PK). The others are called secondary keys. This is the key constraint. We underline the PK in the relation schema.

In an instance $r(R)$, the value of the PK within any tuple cannot be `NULL`. If the PK is composed of many attributes, we do not allow any of the attributes to be `NULL`. This is the entity integrity constraint.

For a relationship from the referencing relation R_1 to the referenced relation R_2, there has to exist an attribute (or a set of attributes), called the Foreign Key (FK), in R_1 that either has the same value with the PK in R_2 or is `NULL`. That is,

$$t_1[FK] \text{ references } t_2[PK] \implies t_2[PK] = t_1[FK] \text{ or } t_1[FK] = \texttt{NULL}.$$

So, the value of the foreign key, if not null, must be an existing primary key. Note that this is one-directional. It is still possible for us to have a primary key whose value is never a foreign key. This is denoted by a directed arc from $R1.FK$ to $R2.PK$.

1.3 Functional Dependency

Functional Dependency Theory is used to quantify the degree of goodness of a relational schema. Goodness is in terms of the attributes, e.g. PK and FK. To quantify the degree of goodness, we introduce the normal forms. We also need to judge whether there are pitfalls to the relational model. The relational schema should also be efficient in terms of the insertion, deletion and update operations. We should convey as much information as possible while minimising redundancy (repetition of data and uncertainities).

Informal guidelines for relational schema

There are 4 informal guidelines used to judge a relational schema, that we will consider below.

The first guideline is that the attributes of a relation should make sense. So, attributes of different entities, e.g. students and courses should not be stored within the same relation completely. Our objective is to minimise redundancy between relations, so we should store different entities separately.

Any relationship between relations should only be represented through Foreign Keys and Primary Keys, e.g. if we have a student taking a set of courses, we should have 3 relations- one for students, one for courses, and one for their relationship

`STUDENT_COURSE(GUID, COURSE_ID, TUTORIAL_GROUP),`

where `GUID` and `COURSE_ID` are primary keys of the relations student and course respectively.

The second guideline is that we should avoid redundant tuples. Having repeated tuples means:

- the storage cost increases, since we need to store redundant tuples.
- the inconsistency cost and operation anomalies also increases, because we the replicas need to be kept consistent during insertion, deletion and update of tuples.
- the replicas might result in consistency anomalies.

When we delete a tuple, we need to delete other tuples that depend on this tuple- we cascade the deletion operation. When we update a tuple, we need to update other tuples similarly- we propagate the update operation. We must do this to maintain consistency within the schema.

Assume that we have a relation schema containing only the following relation:

`EMP_PROJ(SSN, PNumber, Hours, EName, PName, PLocation).`

This relation stores the number of hours an employee is working on a project per week. Within an instance of this relation, we expect there to be a repetition of many attributes within many tuples, e.g. if an employee is working on many projects.

Now, assume that we have 600 employees working on the project `PNumber = 2` with `PName = ProductY`. We want to change the name of the project

ProductY to ProductZ. To enforce consistency, we need to change all the 600 tuples. This is very inefficient for such a small change. If we didn't do this update, then some employees would be working on an inexistent project. Ideally, we should only have to update one tuple.

Next, if we wanted to delete the project with PNumber = 2, then we would need to delete all the tuples with PNumber = 2. However, we might be removing some employees from the relation by doing so. In the business logic, it might be the case that they get assigned to a default project- we should not have to delete the tuples.

The third guideline is that relations should have as few NULL values as possible. We might have NULL values if:

- the value is not applicable or invalid;
- a value is not known; or
- a value is not available.

Attributes that are frequently NULL should be placed in separate relations to avoid wasting storage and reducing uncertainty.

For example, consider the following relation:

EMPLOYEE(SSN, Name, Phone1, Phone2, Phone3)

In a particular instance of the relation, we might find that only 10 of the 1600 employees have three contact numbers. So, we have 1590 null values in that column, which wastes a lot of storage. We should create another relation for the third contact number: PHONE3(SSN, Phone3), along with the original relation EMPLOYEE(SSN, Name, Phone1, Phone2). Then, we only need to store 10 tuples in the Phone3 relation, which saves a lot of storage.

The fourth guideline is that we should design relations to avoid fictitious tuples after join. A fictitious tuple is one that was not present in the original relation. We illustrate this with an example. So, consider the relation

EMP_PROJ(SSN, PNumber, Hours, EName, PName, PLocation)

As we saw before, this is a bad relation with respect to delete and update anomalies. So, we can break it into 2 relations:

R1(SSN, Ename, PNumber, PName, PLocation)
R2(Hours, PLocation)

The shared attribute for the 2 relations is PLocation. To retrieve information from the two relations, we need to join them with respect to PLocation. In particular, assume that we want to find the working hours per week for each employee. To do this, we would need to join R1 and R2 on the common attribute PLocation. This is an issue, since it creates tuples which did not exist in EMP_PROJ.

We illustrate this with an example. So, we have 3 relations:

R(SSN, PName, PLocation, Hours)
Q(SSN, PName, Hours)
P(PLocation, Hours)

Assume that we have 3 instances of `R`, given below:

SSN	PName	PLocation	Hours
111	PR1	Glasgow	20
222	PR1	Glasgow	10
333	PR2	Edinburgh	23

Then, we will have 3 instances of `Q` and `P`, given by

SSN	PName	PLocation
111	PR1	Glasgow
222	PR1	Glasgow
333	PR2	Edinburgh

PLocation	Hours
Glasgow	20
Glasgow	10
Edinburgh	23

So, when we join the two tables `Q` and `P` with respect to the attribute `PLocation` we get the following table:

SSN	PName	PLocation	Hours
111	**PR1**	**Glasgow**	**10**
111	PR1	Glasgow	20
222	PR1	Glasgow	10
222	**PR1**	**Glasgow**	**20**
333	PR2	Edinburgh	23

We have ended up with 2 fictitious tuples here, which are highlighted in bold. So, we need to choose the right common attribute in order to avoid fictitious tuples.

Formal guidelines for relational schema

Functional Dependency (FD) is a formal metric to measure the goodness of a relational schema. In particular, FD is a constraint derived from the relationship between attributes.

Given a relation, we say that the attribute X functionally determines the attribute Y if a value of X determines a unique value for Y. So, the value of Y can be reconstructed given the value of X. We denote $X \to Y$ to mean that X (uniquely) determines Y. Mathematically, we say that if two tuples have the same value for attribute X, then they have the same value for attribute Y. That is, for tuples t_1, t_2,

$$t_1[X] = t_2[X] \implies t_1[Y] = t_2[Y].$$

This is a constraint on all the instances of R.

We illustrate this with an example. Consider the relation

`EMP_PROJ(SSN, PNumber, Hours, EName, PName, PLocation).`

The following is an instance of the relation.

SSN	PNumber	Hours	EName	PName	PLocation
1	5	10	Chris	PX	G12
2	5	30	Stella	PX	G12
1	7	15	Chris	PY	G45

Here, `SSN` determines `EName`. In the first and the third tuple, the `SSN` values match and so the `ENmae` values also match. So, we have a FD `SSN` $\rightarrow$ `EName`. That is, social security number determines the name of an employee, which makes sense.

Moreover, `PNumber` determines `PName`. In the first and the second tuple, the `PNumber` values match and so the `PName` values also match. So, we have FD `PNumber` $\rightarrow$ `PName`.

There is also another FD. Since $\{$`SSN`, `PNumber`$\}$ is a primary key, these two attributes have to determine all the other attributes. So, $\{$`SSN`, `PNumber`$\} \rightarrow \{$`Hours`, `EName`, `PName`, `Location`$\}$.

We will now look at some properties of FDs:

- If K is a candidate key, then K functionally determines all attributes in R, i.e. $K \rightarrow R$. The converse is also true.
- If $Y \subseteq X$, then $X \rightarrow Y$, e.g. if $X = \{$`SSN`, `EName`$\}$, then

  $$X \rightarrow \{\texttt{SSN}\}, \qquad X \rightarrow \{\texttt{EName}\}, \qquad X \rightarrow X.$$

  This is called reflexivity.
- If $X \rightarrow Y$, then $X \cup Z \rightarrow Y \cup Z$. This is called augmentation.
- If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$. This is called transitivity.

1.4 Normalisation Theory

The theory of normalisation is the progressive decomposition of unsatisfactory (bad) relations by breaking up their attributes into smaller, better relations. We will exploit the FDs to specify which attributes can be PKs and FKs. This is the normalisation theory. We do this by:

- asserting which are the FDs among the attributes using the given specification;
- creating a (big) relation with all the attributes;
- recursively decompose the relation based on FDs into many smaller subrelations such that when we re-join them, it guarantees that no information is lost and reconstructs the big relation without any fictitious tuples.

There are different degrees of decomposition, referred to as Normal Form (NF):

- First normal form (1NF),
- Second normal form (2NF),
- Third normal form (3NF), and
- Boyce-Codd normal form (BCNF).

The higher the normal form, the better optimised the schema is.

A prime attribute is an attribute that belongs to some candidate key of the relation, and a non-prime attribute is not a prime attribute, i.e. it is not a member of any candidate key. For example, in the relation

`EMP_PROJ(SSN, PNumber, Hours, EName, PName, PLocation)`

the attributes `SSN` and `PNumber` are prime attributes, while `Hours`, `EName`, `PName`, `PLocation` are non-prime attributes.

1NF

In the first normal form, the domain D_i of each attribute A_i in a relation R refers only to atomic (simple, indivisible) values. So, 1NF disallows nested and multi-valued attributes. We expect there to be redundant and repeated values.

2NF

In 2NF, we will break the record into multiple records to reduce redundancy. For this, we define full and partial FD. A full FD $X \to Y$ means that if we remove a prime attribute A from the primary key X, then $X \setminus \{A\}$ does not functionally determine Y anymore. That is,

$$X \setminus \{A\} \not\to Y.$$

Otherwise, we say that the FD $X \to Y$ is partial. For example, in the relation

EMP_PROJ(SSN, PNumber, Hours, EName, PName, PLocation),

{SSN, PNumber} → Hours is a full FD. This is because neither SSN → Hours or PNumber → Hours is true. But, {SSN, PNumber} → EName is a partial FD since SSN → EName is true.

We say that a relation R is in 2NF if it is in 1NF and every non-prime attribute A in R is fully functionally dependent on the primary key of R. So, to transform a relation from 1NF to 2NF, we remove all the prime attributes from the primary key that cause partial dependencies. To do this,

- we identify all the partial FDs in the original relation (that is in 1NF);
- for each partial FD, we create a new relation such that all non-prime attributes in there are fully functionally dependent on the new primary key, i.e. the prime attribute in the original relation causing partial FDs.

Then, the new relation(s) will be in 2NF.

3NF

Consider the record below.

SSN	EName	BYear	Address	DNumber	DName	DMgr_SSN
1	Chris	1970	A1	3	SoCS	12
2	Stella	1988	A2	3	SoCS	12
3	Philip	2001	A3	3	SoCS	12
4	John	1966	A4	3	SoCS	12
5	Chris	1955	A5	3	SoCS	12
6	Anna	1999	A6	4	Maths	44
7	Thalia	2006	A7	4	Maths	44

This record is in 1NF- no entry contains more than one value. Moreover, the relation is in 2NF- since there is only one prime attribute, it must fully determine all the non-prime attributes. But the relation is still redundant. The last 3 attributes is the same in the first 5 and the last 2 rows. We could break this into 2 relations to avoid this redundancy. This will transform the record into 2 records, both of which are in 3NF.

If $X \to Z$ and $Z \to Y$, then $X \to Y$ by the transitivity property of FD. In particular, if X is a prime attribute and Z and Y are non-prime attributes, then we say that Y is transitively dependent on X via Z. For example, In the record

{CourseID, Lecturer, School},

we have FD1: CourseID → Lecturer and FD2: Lecturer → School. Moreover, we have FD3: CourseID → School via Lecturer, which is a non-prime attribute.

We say that a record R is in 3NF if it is in 2NF and if there is no non-prime attribute which is transitively dependent on the primary key. So, all non-prime attributes are directly dependent on the PK.

Generalised 3NF

In 3NF, if $X \to Z$ and $Z \to Y$ and X is a primary key, we only consider this to be a problem if Z is a non-prime attribute. However, this still allows for some redundancy. For example, consider the following relation

`EMPLOYEE(SSN, PassportNo, Salary).`

Then, we have the following FDs:

- `SSN` $\to$ `PassportNo`;
- `PassportNo` $\to$ `Salary`;
- `SSN` $\to$ `Salary`.

The final FD is not a transitive FD since `PassportNo` is a prime attribute.

This gives rise to the generalised 3NF. Here, a non-prime attribute A in relation R is fully functionally dependent on every candidate key in R, and is non-transitively dependent on every candidate key in R. We still need to deal with prime attributes.

BCNF

Boyce-Codd Normal Form (BCNF) removes all inherent dependencies. Any attribute should be functionally dependent only on the PK. That is, a relation is in BCNF if whenever $X \to A$, then X is the PK.

For example, consider the relation

`TEACH(Student, Course, Instructor)`

with FDs: $\{$`Student`, `Course`$\}$ $\to$ `Instructor` and `Instructor` $\to$ `Course`. This relation satisfies 3NF since we cannot find a transitive FD from a prime attribute to a non-prime attribute. This is because there is only one non-prime attribute. Still, the relation is not in BCNF since `Instructor` $\to$ `Course` does not satisfy the BCNF condition.

We need to decompose this record into 2 records. We need to ensure that we do not create fictitious tuples with this decomposition. The BCNF theorem tells us how to do this. If R is a relation not in BCNF with $X \to A$ a FD that violates BCNF, then we can decompose R to the following relations:

- R_1 with attributes: $R \setminus A$, and
- R_2 with attributes: $X \cup A$.

If either R_1 or R_2 is still not in BCNF, we repeat the process.

CHAPTER 1

EXAMPLES- CHAPTER 1

1.2 Relational Model

Example 1.2.1. Consider the following problem:

> A company is organised into departments. Each department has a unique number, and a particular employee who manages the department. We keep track of the start date that the employee began managing the department. A department may have several locations. A department controls a number of projects, each of which has a unique name, number and a single location. The company stores the information about the employee (e.g. name, salary, birth date, etc.) and the unique social security number (SSN). An employee is assigned only to one department, but may work on several projects, which are not necessarily controlled by the same department. We keep track of the current hours per week that an employee works on each project and their direct supervisor, who is another employee. The company keeps track of the dependent (e.g. child) of each employee for insurance purposes and the corresponding relationship with the employee (e.g. son, daughter).

Create a relational model for the description above.

From the description, we find that:

- the schema is the company;
- the entities are: department, employee, project and dependent; and
- we have the following relationships:
  - an employee manages a department;
  - a department may have several locations;
  - a department controls a number of projects;
  - an employee is assigned to one department;
  - an employee may work on several projects which are not necessarily controlled by the same department;
  - a department controls several projects;
  - an employee is supervised by a supervisor, who is another employee;
  - an employee has several dependents, each one corresponding to a specific relationship with the employee.

Using this, we get the following relations:

- Employee(Name, SSN, BDate, Address, Salary, Supervisor, Department),
- Department(Name, Number, Manager, ManagerStartDate),
- Dept_Location(Department, Location),
- Project(Name, Number, Location, Department),
- Works_In(Employee, Project, Hours),
- Dependent(Employee, Name, BDate, Relationship).

The relations Employee, Department, Project and Dependent come from the identified entities. We have the relations Dept_Location and Works_In to represent relationships. The underlined attributes are primary keys and the dotted attributes are foreign keys.

1.3 Functional Dependency

Example 1.3.1. Assume we have the following relational schema.

- `Reader(Name, Age, Book Title, ID)`,
- `Book(Title, Subject, Number of Pages, ISBN)`.

Adapt the schema so that we obey relational constraints.

The PK of the reader record is the `ID`, and the PK of book is `ISBN`. The two relations are related by the `title` attribute. However, since title is not unique, it should be related by the `ISBN` attribute. Moreover, this relationship is going from `Book` to `Reader`- a person cannot read a book with ISBN that doesn't exist. So, the records should actually be the following:

- `Reader(Name, Age, ISBN, ID)`,
- `Book(Title, Subject, Number of Pages, ISBN)`.

This obeys the relational constraints.

Example 1.3.2. Assume that we have the following relation.

$$R(B, O, I, S, Q, D)$$

We have the following FDs.

- FD1: $S \to D$
- FD2: $I \to B$
- FD3: $\{I, S\} \to Q$
- FD4: $B \to O$.

Show that $\{I, S\}$ is a candidate key but $\{I, B\}$ is not.

- We show that we have the $\{I, S\}$ is a candidate key. We know that $I \to B$ and $B \to O$. So, transitivity tells us that $I \to O$. Therefore, $I \to \{B, O\}$. Moreover, $S \to D$, so $\{I, S\} \to \{B, O, D\}$. Since we also have $\{I, S\} \to Q$, we find that $\{I, S\} \to \{B, O, Q, D\}$. So, we have shown that $\{I, S\}$ determines all the attributes in the relation. This means that $\{I, S\}$ is a candidate key.
- Now, we show that $\{I, B\}$ cannot be a candidate key. We know that $I \to B$ and $B \to O$, so $I \to O$. So, $\{I, B\} \to \{I, B, O\}$. We cannot add any further attribute into the set using the FDs given above- D, Q and S are not dependent on $\{I, B\}$. So, $\{I, B\}$ cannot be a candidate key.

1.4 Normalisation Theory

Example 1.4.1. Consider the Department relation below.

DName	DNumber	DManager	DLocations
Research	5	333445555	{Beltaire, Sugarland, Houston}
Administration	4	987654321	{Stattford}
Headquarters	1	888866555	{Houston}

Convert this relation into 1NF.

This relation is not in 1NF since the `DLocations` attribute is multi-valued. We normalise it by introducing a tuple for each of the value present, i.e. the following record:

DName	DNumber	DManager	DLocations
Research	5	333445555	Beltaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stattford
Headquarters	1	888866555	Houston

Example 1.4.2. Consider the following relation.

`EMP_PROJ(SSN, PNumber, Hours, EName, PName, PLocation)`

Normalise it to 2NF.

This relation is in 1NF since every attribute is single-valued. We have the following FDs:

- {`SSN`, `PNumber`} → `Hours` is a full FD;
- {`SSN`, `PNumber`} → `EName` is a partial FD with `SSN` → `EName` a full FD;
- {`SSN`, `PNumber`} → `PName` is a partial FD with `PNumber` → `PName` a full FD;
- {`SSN`, `PNumber`} → `PLocation` is a partial FD with `PNumber` → `PLocation` a full FD;

So, we will create 3 relations:

`R1(SSN, PNumber, Hours)`
`R2(SSN, EName)`
`R3(PNumber, PName, PLocation)`

Now, each of the three relations are in 2NF- each non-prime attribute fully depends on the primary key.

Example 1.4.3. Assume that we have the following relation.

`EMP_DEPT(SSN, EName, BYear, Address, DNumber, DName, DMgr_SSN)`

Normalise it to 3NF.

The relation is in 1NF since every attribute is single-valued. Moreover, since the primary key is a single attribute, the relation is in 1NF. Also, the following are the FDs.

- the FD `SSN` $\rightarrow$ `DMgr_SSN` is a transitive dependency via the non-prime attribute `DNumber`;
- the FD `SSN` $\rightarrow$ `DName` is a transitive dependency via the non-prime attribute `DNumber`;
- the FD `SSN` $\rightarrow$ `EName` is a direct dependency since there is no non-prime attribute that determines `EName`;
- the FD `SSN` $\rightarrow$ `BYear` is a direct dependency since there is no non-prime attribute that determines `BYear`;
- the FD `SSN` $\rightarrow$ `Address` is a direct dependency since there is no non-prime attribute that determines `Address`.

We transform a record in 2NF to a record in 3NF by splitting it into the non-prime transitive attribute and the other attributes. So, we split this record into the 2 records:

`R1(SSN, EName, BYear, Address, DNumber)`
`R2(DNumber, DName, DMgr_SSN).`

By storing `DNumber` in `R1`, we can join the two records correctly. It is a foreign key. The two relations are in 3NF.

Example 1.4.4. Consider the following instance of the `TEACH` relation.

Student	Course	Instructor
Naranyan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Zelaya	Database	Navathe

Normalise it to BCNF.

This relation is in 1NF since the attributes are single-valued. Moreover, the FDs are:

- {`Student`, `Course`} $\rightarrow$ `Instructor`
- `Instructor` $\rightarrow$ `Course`

So, the relation is in 2NF and 3NF, but not in BCNF. The violating attribute here is `Instructor` $\rightarrow$ `Course`, so $X = \{$`Instructor`$\}$ and $A = \{$`Course`$\}$. By the BCNF theorem, we know that we should create the relations:

```
R1(Student, Instructor)
R2(Instructor, Course).
```

This gives us the following records:

Student	Instructor
Naranyan	Mark
Smith	Navathe
Smith	Ammar
Smith	Schulman
Wallace	Mark
Wallace	Ahamad
Zelaya	Navathe

Course	Instructor
Database	Mark
Database	Navathe
Operating Systems	Ammar
Theory	Schulman
Operating Systems	Ahamad

Joining these two records will give us back the original record.

CHAPTER 2

SQL

2.1 Creating schema and tables

Structured Query Language (SQL) transforms a relational schema into a machine comprehensible format so that we can perform operations on an instance of the schema automatically. It is a declarative language. So, we declare what to do rather than how to do it. It is different from procedural languages, such as Java, Python and C.

We can create a schema using the create command, e.g.

```
CREATE SCHEMA Company;
```

Every statement in SQL ends with a semicolon. After creating a schema, we create tables using a similar command.

The attributes in a table have different domains. These include:

- numeric data types, such as:
  - integer numbers `INT`
  - floating point numbers `REAL` or `DECIMAL(m, n)` (`n` digits after the decimal and `m` digits before the decimal).
- character/string data types, such as:
  - fixed length of characters with length `n` `CHAR(n)`
  - variable length of characters with maximum length `n` `VARCHAR(n)`
- bit string data types, such as:
  - fixed length bit strings `BIT(n)`
  - variable length bit strings: `BIT VARYING(n)`
- boolean data type, with values `TRUE`, `FALSE` or `NULL`.
- date data type, e.g. in the format `YYYY-MM-DD`.

The following command will create 3 tables:

```
CREATE TABLE Employee (
    FName           VARCHAR(15)         NOT NULL,
    SSN             CHAR(9)             NOT NULL,
    BDate           DATE,
    Address         VARCHAR(30),
    SUPER_SSN       CHAR(9),
    DNO             INT,
    PRIMARY KEY(SSN)
);
CREATE TABLE Department(
    DName           VARCHAR(15)         NOT NULL,
    DNumber         INT                 NOT NULL,
    Mgr_SSN         CHAR(9),
```

```
    Mgr_Start_Date  DATE,
    PRIMARY KEY(DNumber),
    UNIQUE(DName),
    FOREIGN KEY(Mgr_SSN) REFERENCES Employee(SSN)
);
CREATE TABLE Dept_Locations(
    DNumber         INT                 NOT NULL,
    DLocation       VARCHAR(15)         NOT NULL,
    PRIMARY KEY(DNumber, DLocation),
    FOREIGN KEY(DNumber) REFERENCES Department(DNumber)
);
```

Within the create command, each row specifies the attribute (e.g. `SSN`), the domain (e.g. `CHAR(9)`), and any constraints we have (e.g. `NOT NULL`). At the end, we specify any primary and foreign keys.

We can set a default value for an attribute. We can also allow a value to not be `null`. For example,

```
DNO INT NOT NULL DEFAULT 1;
```

We can have a range constraint (called a check clause), e.g.

```
DNumber INT NOT NULL CHECK(DNumber > 0 AND DNumber < 21);
```

We must declare the key constraint within the create command. That is, the primary key value should be listed in the command, and this value must be unique for each tuple. Moreover, the entity integrity constraint requires that the primary key value must not be `NULL`. When we have multiple candidate keys, we use the `UNIQUE` clause. For referential constraints, we need to associate them with the primary key in the corresponding relation, e.g.

```
FOREIGN KEY (Super_SSN) REFERENCES Employee(SSN);
FOREIGN KEY (Mgr_SSN) REFERENCES Employee(SSN);
```

The database system ensures consistency when we use the keywords `PRIMARY KEY` and `FOREIGN KEY`.

When a tuple is deleted and updated, other tuples that refer to this tuple can either:

- get the value `NULL` (command: `ON UPDATE SET NULL`),
- get a default value (command: `ON UPDATE SET <value>`), or
- get updated themselves (command: `ON UPDATE CASCADE`).

We can use the command explicitly to choose what happens when a tuple gets updated/deleted. A database management system must ensure that deletion/update does not make the relational schema inconsistent- it transforms from one consistent state to another.

Now, consider the following SQL create command.

```
CREATE TABLE Employee (
    ...
    DNO         INT         NOT NULL        DEFAULT 1,
    CONSTRAINT EMPPK PRIMARY KEY (SSN),
    CONSTRAINT EMPSUPERFK
        FOREIGN KEY(DNO) REFERENCES Employee(SSN) ON DELETE
        SET NULL ON UPDATE CASCADE,
    CONSTRAINT EMPDEPTFK
        FOREIGN KEY(DNO) REFERENCES Department(DNumber) ON DELETE
```

```
        SET DEFAULT ON UPDATE CASCADE
);
CREATE TABLE Department (
    ...
    Mgr_SSN     CHAR(9)     NOT NULL        DEFAULT "888665555",
    CONSTRAINT DEPTPK PRIMARY KEY(DNumber),
    CONSTRAINT DEPTSK UNIQUE(DName),
    CONSTRAINT DEPTMGRFK
        FOREIGN KEY (Mgr_SSN) REFERENCES Employee(SSN) ON DELETE
        SET DEFAULT ON UPDATE CASCADE
);
CREATE TABLE Dept_Locations(
    ...
    PRIMARY KEY(DNumber, DLocation),
    FOREIGN KEY(DNumber) REFERENCES DEPARTMENT(DNumber) ON DELETE
    CASCADE ON UPDATE CASCADE
);
```

We have labelled each constraint, e.g. `EMPSUPERKEY`. This is optional, but helps identify them. Moreover, we have set foreign key constraints. For example, when we delete a department tuple, the employee gets assigned to the default department with `DNO = 1`, and the department location gets deleted.

2.2 Select, from, where

We will use select-from-where clause to find tuples that match a criterion. The syntax is:

```
SELECT  <attributes>
FROM    <tables>
WHERE   <condition>;
```

So, we first declare what to retrieve. Then, we say from which tables to retrieve this data. Finally, we state the condition that a tuple must satisfy in order for it to be retrieved. This condition must evaluate to either `TRUE`, `FALSE` or `NULL` (also called `UNKNOWN`). However, we are not stating how to implement this, e.g. how do we search and check if the tuple satisfies this condition.

We will now look at some queries. First, consider the following relational schema.

`Employee(Name, SSN, Address, Salary, Super_SSN, DNO)`
`Department(DName, DNumber, Mgr_SSN, Mgr_Start_Date)`
`Product(PName, PNumber, PLocation, DNum)`

We will look at some commands to retrieve relevant tuples from the relational schema.

- To select the address of employees in the department 4 or their salary is less than 31 000, we have the command:

  ```
  SELECT  Address
  FROM    Employee
  WHERE   DNO = 4 OR salary < 31000;
  ```

- To select the name and address of all employees who work in the Research department, we have the command:

  ```
  SELECT  Name, Address
  FROM    Employee, Department
  WHERE   DName = "Research" AND DNO = DNumber;
  ```

  The condition `DName = "Research"` is called a selection condition, while the condition `DNO = DNumber` is a join condition.

- To select the project number, the controlling department number and the department manager's name for a project located in "Stafford, TX", we have the command:

  ```
  SELECT  PNumber, DNumber, Name
  FROM    Employee, Department, Project
  WHERE   PLocation = "Stafford, TX" AND Mgr_SSN = SSN AND
              DNO = DNumber AND DNum = DNumber;
  ```

- To select the name of an employee and their supervisor, we have the following command:

  ```
  SELECT  E.Name, S.Name
  FROM    Employee AS E, EMPLOYEE AS S
  WHERE   E.Super_SSN = S.SSN;
  ```

  We are using variables `E` (for employee) and `S` (for supervisor) because the relation here is recursive.

- To select everything about the employees working in department number 5, we have the following command:

```
SELECT  *
FROM    Employee
WHERE   DNO = 5;
```

- To select the distinct salary of each employee, we have the following command:

```
SELECT  DISTINCT Salary
FROM    Employee;
```

  The outcome of an SQL query is another relation. It is either a set (with distinct elements), or a multiset (with duplicates). To make it a set, we use the `DISTINCT` command.

- To select all the product numbers for those that involve employees whose name is "John", either as workers or managers of departments controlling these products, we have the following command:

```
(SELECT DISTINCT PNumber
FROM    Employee, Department, Product
WHERE   Name = "John" AND Mgr_SSN = SSN AND DNum = DNumber)
UNION
(SELECT DISTINCT PNumber
FROM    Employee, Department, Product
WHERE   Name = "John" AND DNO = DNumber AND DNum = DNumber);
```

  Here, we have broken the command into 2 commands- one where the employee manages that department, and the other where the employee works for a department that controls the products. Note that the operations `UNION`, `EXCEPT` and `INTERSECT` are only defined in sets.

If we do not include the where clause with one table, we get the original table. Instead, if we have multiple tables, then this returns the Cartesian product of the two tables. This will most likely have fictitious tuples. We should make use of the foreign key constraint to return all the possible valid combinations.

2.3 Three-valued logic

SQL is a three-valued logic. So, there are 3 boolean values: `TRUE` (1), `FALSE` (0) and `UNKNOWN` (0.5). Note that every `NULL` value is different to other `NULL` values. In particular, any value compared to `NULL` evaluates to `UNKNOWN`, e.g.

- `Address = NULL` evaluates to `UNKNOWN`,
- `Address <> NULL` evaluates to `UNKNOWN`,
- `NULL = NULL` evaluates to `UNKNOWN`.

We should instead check using the commands `IS NULL` and `IS NOT NULL`. In a query, a tuple is retrieved if and only if the condition evaluates to `TRUE`. If it evaluates to `UNKNOWN` or `FALSE`, the tuple doesn't get retrieved.

The following is the logic table for the `AND` operation.

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

It can be thought of as the minimum operation, e.g. `TRUE AND UNKNOWN` is `UNKNOWN` since $\min(1, 0.5) = 0.5$. The following is the logic table for the `OR` operation.

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

It can be thought of as the maximum operation. The following is the logic table for the `NOT` operation.

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

It can be thought of as $1 -$ `VALUE`.

We can use the `null` condition to retrieve the names of all employees who do not have supervisors.

```
SELECT Name
FROM Employee
WHERE Super_SSN IS NULL;
```

Note that the following does not work.

```
SELECT Name
FROM Employee
WHERE Super_SSN = NULL;
```

As we discussed above, under this command, every tuple will evaluate to `UNKNOWN`, meaning that we do not retrieve any tuple.

2.4 Nested queries

We can nest queries within queries, i.e. a select, from, where block within the where clause. The reason we can do this is that a query returns a relation. The nested query's output is the input to the outer query's where clause. This involves using one of the three operations: `IN`, `ALL` and `EXISTS`. So, the outer query always depends on the inner query.

In a nested uncorrelated query, the inner query does not depend on the outer query. In that case, we first execute the nested query and then execute the outer query, using the inner query's output. So, it is executed once for a given command.

In a nested correlated query, the inner query also depends on the outer query. Here, we execute the nested query for each tuple of the outer query individually and then execute the outer query. So, it is executed once for each tuple of the outer query.

The operator `IN` checks whether a value belongs to a set (or a mutiset). For example, to select the SSN of those employees working in the product numbers 1, 2 or 3, we have the command:

```
SELECT SSN
FROM Employee, Department, Product
WHERE DNO = DNumber AND DNUm = DNumber AND SSN IN (1, 2, 3);
```

Now, we look at a nested version of the `IN` operator. To select the names of the employees working in the department "Research", we have the command:

```
SELECT Name
FROM Employee
WHERE DNO IN (
    SELECT DNumber
    FROM Department
    WHERE DName = "Research"
);
```

This is a nested uncorrelated query since we do not make use of the actual employee in the inner query.

The operator `ALL` compares a value with all the values from the inner query's output set using one of the following: `>`, `>=`, `<`, `<=`, `=`, `<>`. For example, to select the name of those employees whose salary is greater than the salary of all the employees in Department 5, we have the command:

```
SELECT Name
FROM Employee
WHERE Salary > ALL (
    SELECT Salary
    FROM Employee
    WHERE DNO = 5
);
```

This too is a nested uncorrelated query.

Now, to select the name of each employee who has a dependent with the same first name as the employee, we have the command:

```
SELECT E.Name
FROM Employee AS E
WHERE E.SSN IN (
    SELECT D.ESSN
    FROM Dependent AS D
```

```
    WHERE E.Name = D.Dependent_Name
);
```

This is a nested correlated query. There is a global scope (outer query), and a local scope (inner query). We have to execute the inner query for each employee. In the inner query, we compute the SSN of the employee whose dependent shares the name with the original employee. In the outer query, we check whether the original employee is part of this set. We can write this as an unnested query as well.

```
SELECT Name
FROM Employee AS E, Dependent AS D
WHERE E.SSN = D.ESSN AND E.Name = D.Dependent_Name;
```

In fact, every correlated query using `IN` can be collapsed into a single block of unnested query.

The operator `EXISTS` checks whether the inner query's output is an empty set or not. For example, to select all the employees that are working at some department, we have the command:

```
SELECT E.Name
From EMPLOYEE AS E
WHERE NOT EXISTS (
    SELECT *
    FROM Department AS D
    WHERE E.DNO = D.DNumber
);
```

Now, to select the name of all employees that work in the department Research (as a nested correlated query), we have the following command.

```
SELECT E.Name
FROM Employee AS E
WHERE EXISTS (
    SELECT *
    FROM Department AS D
    WHERE E.DNO = D.DNumber AND D.DName = "Research"
);
```

Next, to select the name of all the employees that manage a department and have dependents, we have the following command.

```
SELECT E.Name
FROM Employee AS E
WHERE EXISTS (
    SELECT *
    FROM Dependent AS P
    WHERE E.SSN = P.ESSN
) AND EXISTS (
    SELECT *
    FROM Department AS D
    WHERE E.SSN = D.MGR_SSN
);
```

Now, consider the following relational schema:

```
Student(Name, StudentID, Class)
Course(Name, CourseID, Credits, School)
Grades(StudentID, CourseID, Grade)
```

We want to retrieve the names of all students who have a grade of "A" in all of their courses. Then, the command would be:

```
SELECT Name
FROM Student
WHERE StudentID NOT IN (
    SELECT S.StudentID
    FROM Student AS S, Grade AS G
    WHERE G.Grade <> "A" AND S.StudentID = G.StudentID
);
```

This is a nested uncorrelated query. Here, the inner query finds all the students for whom there exists a grade that is not A, and we reject a student if they are present in this set. We can also compute it in the following way.

```
SELECT S.Name
FROM Student AS S
WHERE NOT EXISTS (
    SELECT *
    FROM Grade AS G
    WHERE G.Grade <> "A" AND S.StudentID = G.StudentID
);
```

This is a nested correlated query. Here, the inner query finds all the courses for the student where they got a grade that is not A, and we reject a student if this set is non-empty.

2.5 Inner and Outer Joins

When we join two records, we add to the where clause the condition where the PK value of one record matches the FK value of another one. This is called inner join with the equijoin condition, where it is of the form `R1.PK = R2.FK`.

For example, consider the following schema.

```
Employee(Name, SSN, Address, Salary, Super_SSN, DNO)
Department(DName, DNumber, Mgr_SSN, Mgr_Start_Date)
Product(PName, PNumber, PLocation, DNum)
```

If we want to show the employees who are working in the department "Research", we have the following query.

```
SELECT Name, Address
FROM (Employee JOIN Department ON Dno = DNumber)
WHERE DName = "Research";
```

This is equivalent to the query we had seen before.

```
SELECT Name, Address
FROM Employee, Department
WHERE DName = "Research" AND Dno = DNumber;
```

Here, `DName = "Research"` is the selection condition, and `DNo = DNumber` is the equi-join condition.

The value of the FK can also be `null`, and we know that any condition involving `null` gets the value `UNKNOWN`. So, if we use inner join, a tuple is retrieved if and only if there exists a matching tuple, i.e. the FK is not `null`. When we have `null` values in the FK, we need to use outer join.

In left outer join, a tuple on the left relation must appear in the result. If there is no matching tuple on the right relation, we just add `null` values for the attributes. We can similarly define right outer join. For example, if we want to find the name of an employee and the name of their supervisor (if there exists), we have the following query:

```
SELECT E.Name, S.Name
FROM (Employee AS E LEFT OUTER JOIN Employee AS S
    ON E.Super_SSN = S.SSN);
```

The following is the outcome of the query, using the dataset above.

E.Name	S.Name
John	Franklin
Franklin	James
Ramesh	Franklin
Joyce	Franklin
Ahmad	Jennifer
James	NULL

By using the left outer join, we guarantee that the every employee appears on the first column. However, it is possible that the FK value is `NULL`, in which case `S.Name` becomes `NULL`. Now, assume we execute the following query.

```
SELECT E.Name, S.Name
FROM Employee AS E, Employee AS S
WHERE E.Super_SSN = S.SSN;
```

In this case, we would not get the last cell from the table above.

2.6 Aggregation Functions

An aggregation function is a statistical summary or a value over a group of tuples. The following are built-in aggregation functions over an attribute X:

- `COUNT(*)`, which counts the number of tuples present in the query, e.g. the number of employees are working in department 5.
- `SUM(X)`, which adds the values of the attributes from the tuples present in the query, e.g. the sum of all salaries of employees in department 5.
- `MAX(X)` or `MIN(X)`, which returns a single tuple with the minimum/maximum value in the tuple, e.g. the youngest employee in department 5.
- `AVG(X)`, which computes the average of the values of the attributes from the tuples present in the query, e.g. the average salary of employees working in department 5.
- `CORR(X, Y)`, which computes the correlation coefficient between two attributes, e.g. the correlation between the age and salary of employees working in department 5.

Null values are discarded in aggregate functions, except for `COUNT(*)`. It is also possible to define a new aggregate function.

We can use aggregate functions to show the average salary of those employees working in department 5.

```
SELECT AVG(Salary) AS Average_Sal
FROM Employee
WHERE DNO = 5;
```

Group by

We can partition a relation into groups based on grouping attribute. So, we can cluster tuples when they have the same value in the grouping attribute. For example, we can group the employees working on the same department and count it:

```
SELECT DNO, Count(*)
FROM Employees
GROUP BY DNO;
```

The result will contain precisely one tuple for each department number. We must apply an aggregation function to reduce each group to a single tuple.

We will now show the number of employees per department and the average salary for each department.

```
SELECT DNO, COUNT(*), AVG(Salary)
FROM Employee
GROUP BY DNO;
```

We can also show the number of employees per age.

```
SELECT E.AGE, COUNT(*)
FROM Employee AS E
GROUP BY E.Age;
```

We can similarly compute the average salary of the employees with respect to their age.

```
SELECT E.AGE, COUNT(*), AVG(SALARY)
FROM Employee AS E
GROUP BY E.Age;
```

Similarly, we can find out how many employees are working in each project.

```
SELECT P.PName, COUNT(*)
From Project AS P, Works_on AS W
WHERE P.PNumber = W.PNO
GROUP BY P.PName;
```

Having

The having condition selects a group, i.e. it applies to aggregate functions. For example, assume we have the following relational schema:

Project(PNo, PName)
Works_on(ESSN, PNo).

We can show the number of employees per project, only from those projects with more than 2 employees.

```
SELECT P.PName, COUNT(*)
FROM Project AS P, Works_on AS W
WHERE P.PNo = W.PNo
GROUP BY P.PName
HAVING COUNT(*) > 2;
```

Next, consider the following relational schema:

Employee(Name, SSN, Address, Salary, Super_SSN, DNO)
Dependent(ESSN, Dependent_Name, BDate, Relationship).

We will find the employees (SSN and Name) with more than two dependents and list the number of dependents in the query below.

```
SELECT E.SSN, E.Name, COUNT(*)
FROM Employee AS E, Dependent AS D
WHERE E.SSN = D.ESSN
GROUP BY E.SSN
HAVING COUNT(*) > 2;
```

Now, consider the following relational schema:

Employee(Name, SSN, Address, Salary, Super_SSN, DNO)
Department(DName, DNumber, Mgr_SSN)

We will find the name of the managers of those departments with more than 100 employees.

```
SELECT M.Name
FROM Employee AS M, Department AS D
WHERE M.SSN = D.Mgr_SSN AND D.DNumber IN (
    SELECT E.DNO
    FROM Employee AS E
    GROUP BY E.DNO
    HAVING COUNT(*) > 100
);
```

We first identify the departments with more than 100 employees, and then select managers that manage one of these departments. It is a nested uncorrelated query.

Now, assume that for each department with more than 5 employees, we want to find how many of them make more than £40 000. A naive attempt at this might be the following query.

```
SELECT DNO, COUNT(*)
FROM Employee
WHERE Salary > 40000
GROUP BY DNO
HAVING COUNT(*) > 5;
```

This does not give us the right result. It computes the departments which have more than 5 employees that earn £40 000. We need to apply the condition that the employee earns more than £40 000 after we have found the departments with more than 5 employees. The correct query is the following.

```
SELECT DNO, COUNT(*)
FROM Employee
WHERE Salary > 40000 AND DNo IN (
    SELECT E.DNO
    FROM Employee AS E
    GROUP BY E.DNO
    HAVING COUNT(*) > 5
)
GROUP BY DNO;
```

Next, we want to find the department(s) with the maximum number of employees. It is possible that more than one department has the maximum number of employees. The query is the following. First, we have the following intermediate query.

```
SELECT E.DNO, COUNT(*) AS Members
FROM Employee AS E
GROUP BY E.DNo;
```

At this point, we have the department number, along with a count of the employees working in that department, denoted by the attribute members. Call this record `A`. Using this record, we can find the relevant departments using the following query.

```
SELECT DNO, Members
FROM A
WHERE Members = (
    SELECT MAX(Members)
    FROM A
);
```

We can group the two steps together as follows.

```
SELECT DNO, COUNT(*)
FROM Employee
GROUP BY DNO
HAVING COUNT(*) = (
    SELECT MAX(A.Members)
    FROM (
        SELECT E.DNO, COUNT(*) AS Members
        FROM Employee AS E
        GROUP BY E.DNO
    ) AS A
);
```

CHAPTER 3

DESIGN AND QUERY

3.1 Physical Design

There is a 3-level storage hierarchy- primary storage, main memory such as RAM and cache; secondary storage, such as hard-drive disks (HDD) and solid-state disks (SSD); and tertiary storage, such as optical drives. As we go down the storage hierarchy, the storage capacity increases, but access speed decreases significantly, and the price decreases.

The fundamental challenge is that a database is too large to fit in the main memory. So, we have to store them in secondary storage (i.e. hard disks). This has the following consequences:

- Since the hard disk is not CPU-accessible, we need to first load the data into main memory from the disk, and then process it in the main memory.
- The speed of data access becomes low- data access from HDD takes 30ms, while it only takes about 30ns in RAM.

So, the main bottleneck in query execution is transferring data from the disk to main memory. We will organise data on disks in such a way that minimises this latency.

Our first challenge is therefore to organise tuples on the disk to minimise I/O access cost. We will first consider how data is represented. A tuple is represented as a sequence of binary digits. Records are grouped together into blocks. A file is composed of many blocks. This is illustrated below.

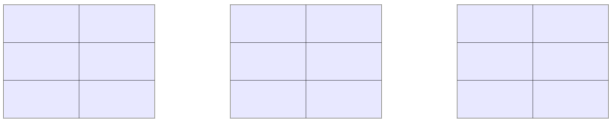


Figure 3.1: A file stored in blocks

Here, the 3 tables form a file; each table represents a block; and a cell in the table is a record.

Records can be of fixed or variable length. This depends on whether the attributes have a fixed size (in bytes), or whether the size of the attributes can vary. We will assume that the attributes have fixed size.

A block is of fixed length. The blocking factor (bfr) is the number of records that can be stored in a block. It is given by

$$bfr = \text{floor}(B/R),$$

where a record occupies R bytes and a block can store B bytes. For example, if $bfr = 100$, then we can store up to 100 records in a single block. We must have at least one record per block, so we require $B \geq R$.

Another challenge is how we allocate the blocks of files on the disk. We can use linked allocation. Here, each block i has a pointer to the physical address to the logically next block $i + 1$ anywhere on the disk. So, it is essentially a linked list of blocks. This is illustrated below.

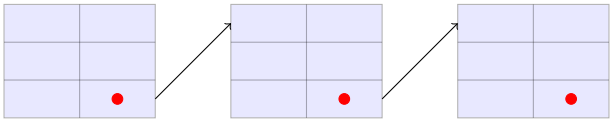


Figure 3.2: A file as a linked list of blocks

Our next challenge is to distribute records within blocks to minimise I/O cost. For this, we will consider 3 representations of data:

- heap (or unordered) file, where we add a new record to the end of the file.
- ordered (or sequential) file, where we keep the records physically sorted with respect to some ordering field.
- hash file, which makes use of a hash function to each record, and uses this as the physical block address.

If we use ordered file, then we need to decide on the ordering field. Similarly, if we use hash file, then we need to choose the hash field. This should be done in a way that minimises the I/O cost.

We define I/O access cost as the cost for:

- retrieving a whole block from the disk to memory to search for a record with respect to some searching field(s) (the search cost); and
- inserting, deleting or updating a record by transferring the whole block from memory to disk (the update cost).

The cost function is the expected number of block accesses (read or write) to search, insert, delete or update a single record. The block is the minimum communication unit. We can only transfer blocks, and not records from disk to memory (and vice versa).

Heap files

Inserting a new record is efficient for heap files. We just load the last block from disk to memory. The address of the last block is part of the file header. Then, we insert the new record at the end of the block and write it back to the disk. We have 2 block accesses every time, so this is $O(1)$ block accesses.

Searching for a record is inefficient. We have to perform a linear search through all the b file blocks. In particular, we load a block at a time from disk to memory and search for the record. If the searched tuple is not unique, we need to access all the files. Assuming that the searched tuple is unique, then it takes about $b/2$ block accesses to find it. In both cases, this is $O(b)$ block accesses. Since databases store large amounts of data, this is considered inefficient.

Similarly, deleting a file is inefficient. We need to first search the records. Then, we remove it from the block and write the block back to the disk. This leaves unused spaces within blocks, which is quite inefficient. So, this is also $O(b)$ block access. To simulate deletion and avoid this inefficiency, we can use deletion markers. In that case, each record stores an extra bit, where the value 1 means that the record has been deleted. We periodically reorganise the file by gathering the non-deleted records and freeing up blocks with deleted records.

Sequential files

All the records in a sequential file are physically sorted by an ordering field. We keep the files sorted at all times. Sequential file storage is suitable for queries that require:

- sequential scanning, i.e. data returned in some order (if it is the same as the ordering field);
- order searching, e.g. data similar to given values; and
- range searching, i.e. values between two values (if it is the same as the ordering field).

Retrieving a record using the ordering field is efficient. The block is found using binary search on the ordering field, so we require $O(\log_2 b)$ block accesses. However, retrieving a record using a non-ordering field is not efficient- it is the same as the heap files. So, we require $O(b)$ block accesses. We cannot exploit the sorted nature of the file.

Range queries (with respect to the ordering field) are efficient for sequential files. We just search for the minimum value, and then keep reading the blocks until we find a value bigger than the maximum value. So, the original search is $O(\log_2 b)$ block accesses, and we just read the blocks until the range is exhausted, which is $O(b)$ block accesses. This cannot be minimised since we want to read precisely these blocks; the efficiency comes from the original binary search.

Insertion is not efficient for sequential files. We first locate the block where the record should be inserted- this requires $O(\log_2 b)$. On average, we will need to move half of the records to make room for the new record. This is very expensive for large files. We can alternatively use chain pointers. Here, each record points to the logically next ordered record. If there is free space in the right block, we insert the new there. Otherwise, we insert the new record in an overflow block and use chain pointers. Pointers must be updated; it is a sorted linked-list. Having pointers increases overflow, and we will have to periodically reorganise the blocks.

Deleting a tuple is expensive as well. First, we have to locate whether the record is to be deleted. We can delete the entry, which would add a gap. Also, we can make use of deletion markers. In that case, we update the deletion marker from 0 to 1, and update the pointer not to point to the deleted record. In both cases, we have to periodically reorganise the blocks.

Updating on the ordering field is expensive. The record is deleted from the old position and gets inserted into its new position- it involves both insertion and deletion, both of which are expensive. However, updating on a

non-ordering field is efficient. We just find the tuple and update one attribute, and write the block back to the disk. So, this involves $O(\log_2 b) + O(1)$ block accesses.

Hash files

In hashing, we partition the records into M buckets. Each bucket can have one or more blocks. We then choose a hashing function $y = h(k)$, with output in $\{0, 1, \ldots, M-1\}$, where k is the value of the attribute in the hash field. For this to work efficiently, we require h to uniformly distribute records into the buckets. That is, for each value k, a bucket is chosen with equal probability $1/M$. An example of such a hashing function is the modulo function, i.e. $y = h(k) = k \bmod M$.

Mapping a record to a bucket $y = h(k)$ is called external hashing over hash-field k. Normally, there is a chance that collisions occur. That is, two or more records are mapped to the same bucket. For example, let $M = 3$ and $h(k) = k \bmod 3$. So, we have 3 buckets. If the attribute values are $k = 1, 11, 2, 4$, then we map them to buckets 0, 2, 2 and 1 respectively. So, there is a collision on bucket 2. This is indirect clustering- we are grouping tuples together with respect to their hashed values y, and not with respect to their hash field values k.

If we want to retrieve a possible record with respect to the hash field, we do the following:

- we hash the hash attribute k and get the corresponding bucket;
- we use the hash map to get the block address in disk of the relevant bucket;
- we fetch the block from the disk to memory; and
- we search the block in memory linearly to find the record.

So, we require $O(1)$ block access.

Due to collisions, a hash bucket might be full. Using the chain pointers method, we can insert a new record hashed to a full bucket. If there are $O(n)$ overflown blocks per bucket, then we require $O(1) + O(n)$ block accesses. To improve efficiency, we should use a hash function that minimises the number of overflow blocks.

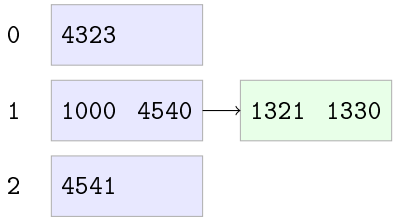
Now, consider deleting a record based on the hash field. We find the record in the main bucket, or an overflown block. So, this takes $O(1) + O(n)$ block accesses. We would need to periodically pack the blocks together to free up the space from deleted records.

When we update a record based on a non-hash field, we locate the record in main or overflow bucket. We load the block into memory, update it and write it back. So, this also takes $O(1) + O(n)$ block accesses. When we update a record on the hash field, we need to delete the record from the old bucket and add it to the new bucket. This also takes $O(1) + O(n)$ block accesses.

Now, assume that $M = 3$, we have 1 block per main bucket, and $bfr = 2$ records per block. The hash attribute is the `SSN` key values:

$$\{1000, 4540, 4541, 4323, 1321, 1330\}.$$

If the hash function is $h(k) = k \bmod 3$, then we can assign the employees into buckets as follows:



The blue blocks are part of the main bucket, while the green ones are the overflow blocks. We require 4 buckets to hash these values. If we were using heap or sequential representation, we would only need 3 blocks. Since the hashing function does not uniformly distribute the blocks, we instead require 4 here.

However, range queries are inefficient in hash representation, even when the range is over the hashed field. For each value in the range, we have to find and retrieve the bucket and select the value from the record. The hash function uniformly distributes the values, so we would not want it to be possible to retrieve all values within the same bucket. So, it takes $O(m) + O(mn)$ bucket accesses where m is the range of the distinct values in the range and $O(n)$ overflow blocks per bucket. This is not efficient. Sequential files are much better in this case.

The distribution of values influences the expected cost. This implies that the expected cost is unpredictable to compute in practice. For instance, assume that we have an age distribution as shown in the table below.

Age	Count
20	2 000
30	900
40	800
50	80
60	23
70	80

Moreover, let $bfr = 40$ employees per block. For people aged 20, we need to retrieve

$$\text{ceil}(2000/40) = 50$$

blocks. Instead, if we want to get people aged 60, we need to only retrieve

$$\text{ceil}(23/40) = 1$$

block. So, the number of block accesses depends on the value.

3.2 Indexing Methodology

We have looked at physical design of the database. The objective there was that, given a specific file type, we provide a primary access path based on a specific searching field. Now, we will look at index design. In this case, our objective is, given any file type, we provide a secondary access path using more than one searching field.

Adding a secondary access path means that we need to store additional (metadata) files on the disk. Moreover, we need to maintain them so that they are consistent with the actual data. Although the overhead has increased, we significantly improve the searching process by avoiding linear scanning.

When we create an index, we need to follow some principles.

- We create one index over one field. This is called the index field. We will only consider indices built on one attribute.
- An index is another separate file. This is a metadata file, and is the reason the overhead increases.
- All index entries are unique and sorted with respect to the index field. We expect a tuple in the relation to be of the form (`index-value`, `block-pointer`). The block pointer is the location of the block that contains the entry with the given value.
- First, we search within the index file to find the block pointer, and then we access the data block from the data file.

For indexing to be more efficient than linear searching, we require the index files to occupy less blocks than the data file. This is true because the index entries are quite small, with only two attributes. So, we can fit in more index entries in a block than data records, i.e. most of the time,

$$bfr(\textit{index-file}) > bfr(\textit{data-file}).$$

There are 2 indexing strategies. A dense index has an index entry for every record in the file, as illustrated below.

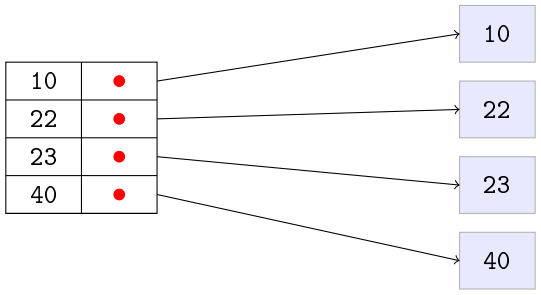


Figure 3.3: A dense index

So, the index file has the same number of entries as the data file. A sparse index has an index entry for only some of the records, as illustrated below.

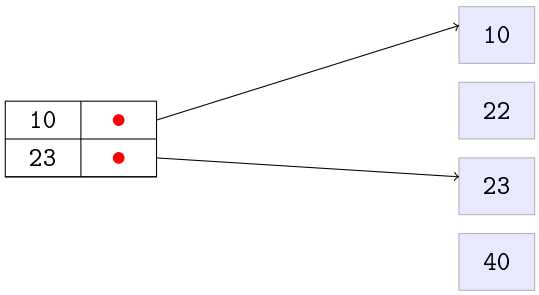


Figure 3.4: A sparse index

This can be used when the indexing field is how the data files are sorted. So, if the value is 22, we look at the block containing 10- the next block starts with 23, so if 22 is present in the data file, it must lie within this block.

Another expectation we have is than searching over index is faster than over the file. Since indexing file is an ordered file, we can adopt a binary/tree-based approach to find the pointer to the actual data block. This makes the procedure much faster than linear search.

There are 3 types of index types- primary, clustering and secondary index. Primary index is where the index field is an ordering, key field of a sequential file (e.g. `SSN`, where the file is sorted by `SSN`). Clustering index is where the index field is an ordering, non-key field of a sequential file (e.g. `DNO`, where the file is sorted by `DNO`). Secondary index is where the index field is non-ordering, i.e. the files are not sorted with respect to this attribute. The index field may be key or non-key.

Primary Index

A primary index is built over a sequential file, where the indexing field is the field the files are sorted by. Since the data file is sorted, we can use sparse indexing to index the data files, as illustrated below.

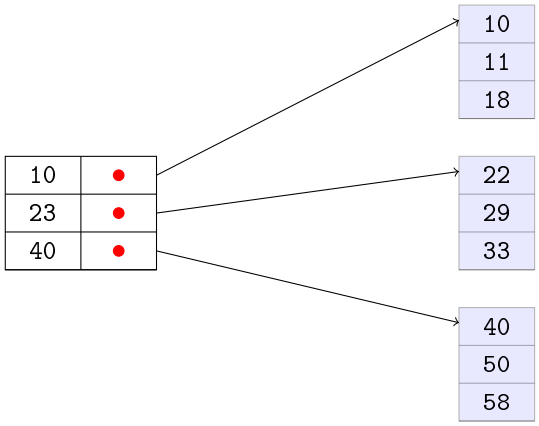


Figure 3.5: Primary Indexing

So, we have one entry in the index file per data block. The i-th index entry (k_i, p_i) refers to the i-th data block. The value k_i is the field value of the first record in block i. The first data-record in block i with value k_i is called the anchor of block i.

To find a particular tuple based on the index entry, we first search in the index file the block where we expect this entry to be. The index entries are sorted, so we use a binary search. This gives us the anchor entry of the block. We then load this block and search within the block for the tuple.

If the anchor record is deleted/updated, we need to update the sequential data file- this is a very costly computation. Moreover, after the update, we need to propagate the update to the index file. The index file is also a sequential file, so this too is a costly computation.

If a non-anchor record is updated, we need to update the sequential data file. This need not be propagated to the index file if the order of the blocks isn't affected. If the order of the blocks is affected, or the non-anchor record is deleted, we need to update the data file and the index file like we did previously.

Clustering index

A clustering index is used to index a sequential file on an ordering, non-key field. The indexing file is a set of cluster of blocks, with a cluster per distinct value. The block pointer points at the first block of the cluster. The other blocks of the same cluster are contiguous and accessed via chain pointers. The clustering index is sparse because we are only using some of the index entries to index the entire data file.

An illustration of clustering index is given below.

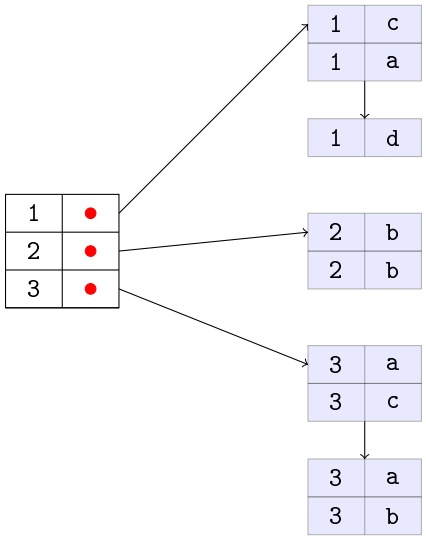


Figure 3.6: Clustering Index

The number of index entries is the same as the number of the clusters. Each pointer in the index entry points to the first entry of the cluster. Each block is pointing to the next contiguous blocks using a linked list data structure.

Secondary Index

Now, we want to index a file on a non-ordering field. The file might be undordered, hashed, or ordered, but not ordered with respect to the indexing field. There are 2 cases here, where the secondary index is unique or not.

If we want a secondary index on a non-ordering, key field, we need one index entry per data record, i.e. a dense index. This is because the entries are scattered around the data file with respect to this index, and so we cannot use anchors.

An illustration of secondary non-ordering key index is given below:

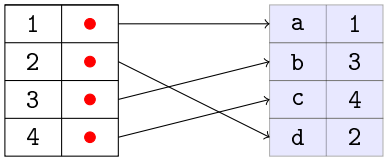


Figure 3.7: Secondary Index on a non-ordering, key index. Note that we are linking to tuples in the figure for simplicity, and not to blocks.

For each index entry, we store the block pointer where the tuple can be found.

The second type of secondary index is on a non-ordering, non-key field. So, some records are having the same indexing value. Moreover, the data files are not sorted with respect to the index attribute. So, we need to store all the block pointers for a given index in yet another file.

This is illustrated in the figure below.

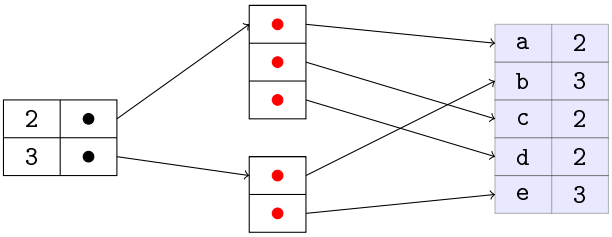


Figure 3.8: Secondary index on a non-ordering, non-key attribute. Note that we are linking to tuples in the figure for simplicity, and not to blocks.

In the figure above, we have 2 possible values for the index- 2 and 3. Within the indexing file, we point to a secondary indexing file. In the secondary index file, we have block pointers for all the data records that have the matching index value (a cluster/group). Indexing at the first level is sparse since there are many tuples in the data field with the index value since this value is not unique.

So, there are 2 levels of indirection here. At the first level, we have a block of block pointers of a cluster. At the second level, a block pointer points to the data block that has records within this distinct index value.

Multi-level index

All the index files (primary, clustering and secondary) can be used to build multi-level index files. They are all ordered with respect to the indexing field. Moreover, the indexing field have unique values, and each index entry is of fixed length. This means we can create an primary index on each of these index files. We could even build an index over that index, and so on. This is multi-level indexing.

The original index file is called the base or Level-1 index file. Indexing this index file gives us a Level-2 index file. We can continue this on to get a Level-t index file for any positive integer t. Now, we aim to find the best level-t of a multi-level index to expedite the search process trading off speed-up with overhead.

The following is an example of a level-2 index.

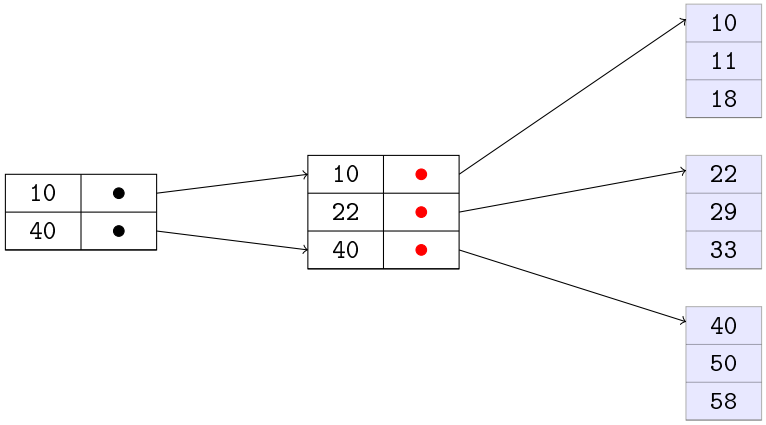


Figure 3.9: A 2-level index

The first index is a primary index, so it is using anchors. Also, the second level is a primary index, so it points to the anchors within the level-1 index file. The level-2 index has 1 block (containing 2 records), so we should not create level-3 index on it.

In general, we should stop not create level-$t+1$ index if level-t where the top-level index has 1 block. So, given a level-1 index with blocking factor m entries/block, the multi-level index is of maximum level

$$t = \log_m b.$$

The value m is known as fan-out.
If we want to find a tuple from a level-t index, we first load the t-level index block. Using this block, we can load a single $t-1$-level block, and so on. We continue this until we load a L1 block. So, we need precisely t block accesses to load the L1 data block. Finally, we load the actual data file block using the L1 index. If it takes n block accesses to retrieve the data block from the L1 block, it will take us

$$t + n = \log_m b + n$$

block accesses to retrieve the relevant tuple(s).

3.3 B and B+ Trees

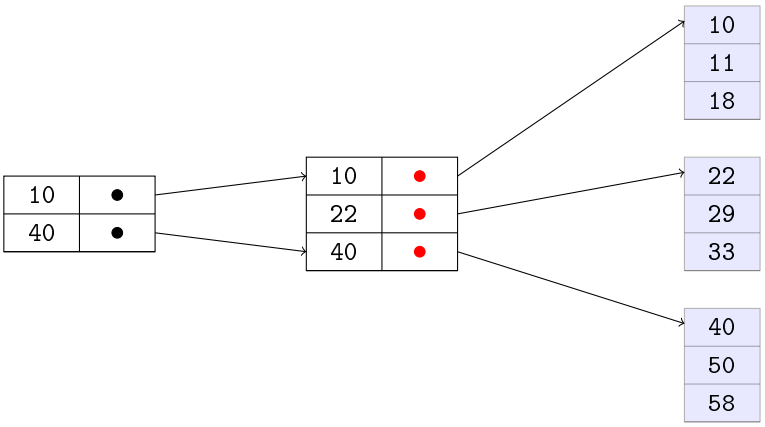
For multi-level indices, we search for a record over a t-level index with $t+1$ block accesses. However, insertions, deletions and updates over the data file are reflected over the multi-level indices. This makes the operations costly. This is because all encapsulated indexes are physically ordered files. So, all updates should be reflected to all levels.

We will try to define a dynamic multi-level indices. This means that the data structure should:

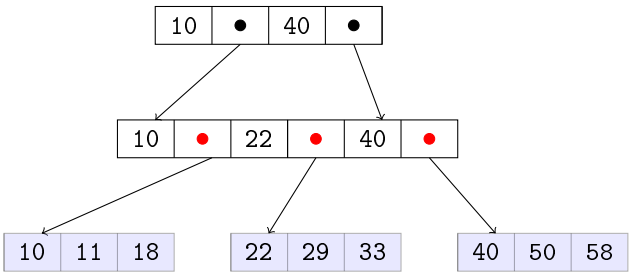
- adjust to deletions and insertions of records;
- expand and shrink following the distribution of the index values; and
- be self-balancing (subtrees should be of the same depth).

We will do this using B and B+ Trees.

We can envisage a multi-level index as a tree. This can be done by turning it clockwise by 90 degrees. For example, consider the following diagram.



We can rotate it clockwise by 90 degrees to get the following tree.



Here, the root is the level-2 index. The children of the root are level-1 index blocks. The leaves are the actual data blocks.

There is a limitation to this. The tree needs to be balanced. This is because it does not adjust to keys' distribution. That is, the leaf nodes are at different levels. The number of block accesses in a multi-level index depends on the

depth of the tree, so we would like the tree to be as balanced as possible. In the worst case, we would end up with a linked list of nodes.

Our first challenge is to ensure that the tree is balanced by minimising the tree depth t. This challenge has two parts- how do we deal with insertion into a full node, and deletion of a node. We can split a node into two if the node is full, and merge two nodes if a node is to be deleted. This increases the overhead, but is required to keep the tree balanced.

B Tree

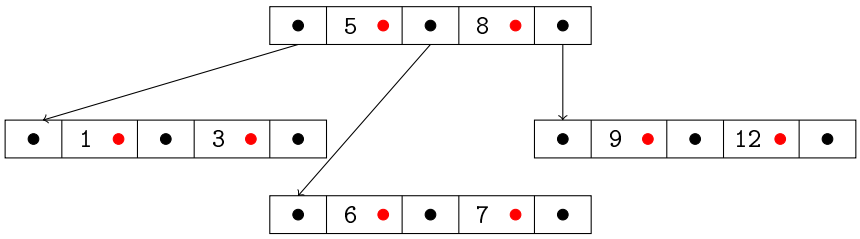
B Trees are used for secondary indices with a key, non-ordering attribute. A B tree node of order p splits the searching space up to p subspaces. We assume $p > 2$. A node is of the form

$$\{P_1, (K_1, Q_1), P_2, (K_2, Q_2), \ldots, P_{p-1}, (K_{p-1}, Q_{p-1}), P_p\},$$

where

- K_i is a key value,
- P_i is a tree pointer for those tuples with $X < K_i$,
- P_{i+1} is a tree pointer for those tuples with $X > K_i$,
- Q_i is a block pointer for K_i.

A pictorial representation of this is given below.



The advantage of B trees is that every time we descend a level, we split the searching space into p subtrees, which exponentially lowers the searching space. If $p = 2$, this is equivalent to binary search. We typically have $p > 2$. Like in the case above, a tree of order p can store up to p tree pointers (shown in black) and up to $p-1$ key values and their block pointers (shown in red). In the case above, we have $p = 3$- 3 tree pointers, 2 key values and data pointers in each node.

When searching for a tuple of a value using a B tree, we start from the root and descend in the right branches until we get to a key value corresponding to the tuple. Then, we access the corresponding data block and return the matching tuple.

Normally, a data block contains all the elements of the tree. So, each tree node corresponds to a block. We now consider the number of block accesses we need to find the right tuple with the B tree given above:

- If we search for 8, we first access the root block. Since this node contains the value 8, we can access the data block containing 8 with another block access. In total, we need 2 block accesses.
- If we search for 7, we first access the root block. This node does not contain 7, so we descend the middle branch. This block does contain the value 7, so we can access the data block containing 7 with another block access. In total, we need 3 block accesses.
- If we search for 31, we first access the root block. This node does not contain 7, so we descend the root branch. This block also doesn't contain 31, but we know there is no right tree pointer here. This means that there is no record corresponding to the value 31. So, we just need 2 block accesses.

B+ Trees

A B Tree gives rise to a big metadata. We need to find another data structure so that it has the advantages of B tree while minimising storage. A B tree node stores a lot of data: tree pointers, data pointers and key values. We need tree pointers and key values to navigate the B tree.

We want to be storage efficient by freeing up space from the nodes. Also, we want to be search efficient by maximising the fan out (the splitting factor) of a node. We can increase the order of the node to increase fan out. To do this, we get rid of data pointers from the nodes. We still need to store data pointers, but we can remove them from non-leaf nodes.

In a B+ Tree, we have 2 types of nodes. An internal node guides the search process, and only have tree pointers and key values. A leaf node points to actual data blocks. To maximise fan out, internal nodes have no data pointers. However, we need to go to the leaf to access any data pointers. Nonetheless, since we have removed data pointers from the nodes, this process is quite fast.

Moreover, the leaf nodes hold all the key values sorted and their corresponding data pointers. Also, some key values are replicated in the internal nodes to guide and expedite the search process. This corresponds to medians of key values in sub-trees.

A B+ internal node of order p is of the form

$$\{P_1, K_1, P_2, K_2, \ldots, P_{p-1}, K_{p-1}, P_p\},$$

where

- K_i is a key value,
- P_i is a tree pointer for those tuples with $X < K_i$,
- P_{i+1} is a tree pointer for those tuples with $X > K_i$.

A B+ tree of order p can store up to p tree pointers and up to $p-1$ key values.

A B+ internal node of order p_L is of the form

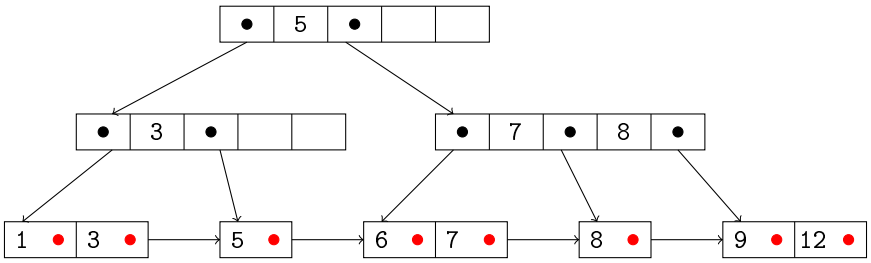
$$\{(K_1, Q_1), (K_2, Q_2), \ldots, (K_{p_L}, Q_{p_L}), P_{\text{next}}\},$$

where

- K_i is a key value,
- Q_i is a data pointer for K_i,
- P_{next} points to the next block of leaf nodes.

We store a linked list of leaf nodes. All the leaf nodes are at the same level, meaning that the tree is balanced.

An example of a B+ tree is given below.



Here, the internal nodes have order $p = 3$ and leaf nodes have order $p_L = 2$.

The leaf nodes are linked and sorted by key. All keys of the file appear at the leaf nodes. The leaf nodes contain data pointers only, to expedite navigation. Leaf nodes are balanced for constant I/O cost. Some selected keys are replicated in the internal nodes for efficiency.

To find the block containing a given value, we must always descend to the leaf nodes. So, it will always take 3 block accesses to find the relevant tuple, assuming it is present.

For a B tree, we saw that we can store 65 535 key values and data pointers in the tree in a very similar scenario. Using a B+ tree, we have almost multiplied the capacity by 4. However, we require about 3 times more blocks for a B tree. Nonetheless, adding a new tuple to the B tree can be costly (e.g. if we add a 65 536th tuple). This is not the case for B+ trees- we might not even have to add the new entry to an internal block. Moreover, adding to the leaf nodes is simple since it is a linked list, even if we need to keep it sorted.

CHAPTER 3

EXAMPLES- CHAPTER 3

3.1 Physical Design

Example 3.1.1. Assume we have the relation `Employee` with $r = 1103$ tuples, each one corresponding to a fixed-length record. Moreover, each record has the following fields:

- `Name` (30 bytes),
- `SSN` (10 bytes), and
- `ADDRESS` (60 bytes).

Given that the block size $B = 512$ bytes, compute the number of blocks we need to fit all the tuples.

A tuple occupies

$$R = 30 + 10 + 60 = 100$$

bytes. So, the blocking factor

$$bfr = \text{floor}(B/R) = 5.$$

This means that we can store at most 5 tuples per a block. So, to store all $r = 1103$ tuples, we need

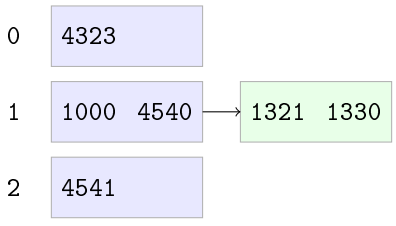
$$b = \text{ceil}(r/bfr) = 221$$

blocks.

Example 3.1.2. Assume we have the relation `Employee` with $r = 6$ tuples with $bfr = 2$. Compute the number of block accesses we need based on different data representations to execute the following query.

```
SELECT  *
FROM    Employee
WHERE   SSN = x;
```

The following is the hash representation of the file.



- First, we consider the hash file representation. We assume that each bucket has equal probability.
  - If the hashed value is 0, then we need to access 1 block in both the best, the average and the worst case.
  - If the hashed value is 1, then we need to access 2 blocks in the worst case and 1 block in the worst case. In the average case, the number of block accesses is
$$\frac{1+2}{2} = 1.5$$
  - If the hashed value is 2, then we need to access 1 block in both the best, the average and the worst case.

  So, the expected block access in the worst case is
$$\frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 2 + \frac{1}{3} \cdot 1 = \frac{4}{3}.$$
  In the best case, the expected block access is
$$\frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 1 = 1.$$
  The expected block access in the average case is
$$\frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 1.5 + \frac{1}{3} \cdot 1 = 1.17$$
  block accesses.
- If we use the heap file structure, we need 3 blocks to store the 6 tuples. So, in the worst case, we have 3 block accesses. In the best case, we just need 1 block access. The expected block access for heap representation in the average case is
$$\frac{1+3}{2} = 2$$
  block accesses.
- In the sequential file structure, we also have 3 blocks. In the average and the worst case, we have
$$\log_2 3 \approx 1.58$$
  block accesses. In the best case, we just need 1 block access.

Example 3.1.3. Assume that we have hash, heap and sorted file where:

- we have 3 blocks in heap and sorted file;
- we have 4 blocks in the hash file- 3 main blocks and 1 overflown block.

Also, let k be the hash attribute, and let $p\%$ be the proportion of the queries that involve the attribute k. Find the best data structure (in the worst case) to store the values with respect to the ratio p.

- If the file is a heap we would need 3 block accesses in the worst case. The attribute k does not matter because the search is always linear.
- Next, assume that the file is sorted with respect to k. If the search uses the attribute k, then we can use a binary search algorithm to get the result in $O(\log_2 b)$ block accesses. Instead, if it does not use the attribute k, then we have to use the linear search in $O(b)$ block accesses. So, the expected number of block accesses in the worst case is

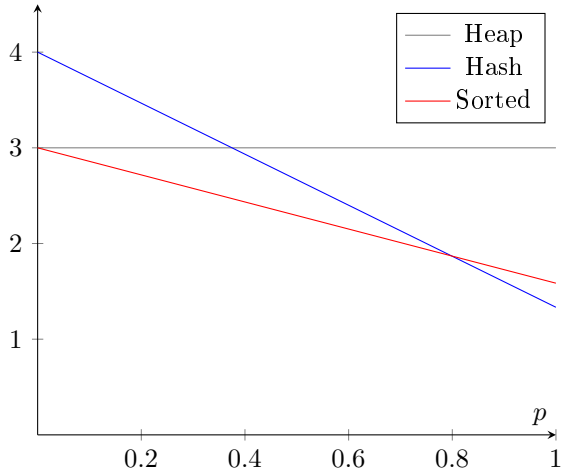
$$p \cdot (\log_2 3) + (1 - p) \cdot 3 = 3 + p \cdot (\log_2 3 - 3).$$

- Now, if we use hash representation, we have

$$p \cdot (\tfrac{1}{3} \cdot 1 + \tfrac{1}{3} \cdot 2 + \tfrac{1}{3} \cdot 1) + (1 - p) \cdot 3 = 4 - \tfrac{8}{3}p$$

  block accesses in the worst case.

We can plot these lines and see which option is better depending on the value of p.



So, if $p < 0.8$, we should use the sorted representation. Instead, if $p > 0.8$, we should use the hash representation.

3.2 Indexing Methodology

Example 3.2.1. Assume we have the `EMPLOYEE` table with key attribute `SSN`.

- We have $r = 300\ 000$ records.
- A record takes $R = 100$ bytes.
- The block size is $B = 4\ 096$ bytes.
- The attribute `SSN` occupies 9 bytes.
- A pointer occupies 6 bytes.
- The data files are sorted with respect to `SSN`.

We want to execute the query.

```
SELECT   *
FROM     EMPLOYEE
WHERE    SSN = k;
```

Compute the number of block accesses we need to execute this query using a linear search, binary search and a primary index.

The blocking factor is

$$bfr = \text{floor}(B/R) = \text{floor}(4096/100) = 40.$$

So, the number of data blocks we need is

$$b = \text{ceil}(r/bfr) = 7500.$$

- A linear search takes on average

  $$b/2 = 3750$$

  block accesses.

- A binary search takes on average

  $$\log_2 b \approx 11.9$$

  block accesses.

- The index entry has components: `(SSN, Pointer)`. We know that we need 9 bytes for SSN and 6 for the pointer, so we need 15 bytes for each index. Therefore, the indexing blocking factor is

  $$ibfr = \text{floor}(4096/15) = 273.$$

  We have 7 500 data blocks, so 7 500 index tuples. So, we need

  $$ib = \text{ceil}(7500/273) = 28$$

  blocks to store all the index files.

  Next, we compute the number of block accesses with the primary index.

– We first search for the SSN in the index. Since the primary index is sorted with respect to the index, it takes about

$$\log_2 28 \approx 4.8$$

block accesses to find it.

– After finding the index block, we need to load the data block. This takes 1 block access.

So, we have 5.8 block accesses to find the relevant tuple.

Example 3.2.2. Assume we have the `EMPLOYEE` table with non-key attribute `DNO`.

- We have $r = 300\ 000$ records.
- A record takes $R = 100$ bytes.
- The block size is $B = 4\ 096$ bytes.
- The attribute `DNO` occupies 9 bytes.
- A pointer occupies 6 bytes.
- The data files are sorted with respect to `DNO`.
- There are 10 departments.
- The `DNO` values are uniformly distributed over the tuples, i.e. there are 30 000 employees in each department.

We want to execute the query

```
SELECT   *
FROM     EMPLOYEE
WHERE    DNO = x;
```

Compute the number of block accesses we need to execute this query using a linear search and a clustering index.

The blocking factor is

$$bfr = \text{floor}(4096/100) = 40.$$

So, the number of blocks we need is

$$b = \text{ceil}(r/bfr) = 7\ 500.$$

- We first consider a linear scan with an exiting feature- we stop as soon as we have found all the tuples. Based on the uniformity assumption, each department has 750 blocks. If we have `DNO = 1`, then we retrieve the first 750 blocks and then stop. If `DNO = 2`, then we retrieve the first two 750 blocks and then stop. Assuming that it is equally likely for the `DNO` to take all these values and

that the value `x` is a valid `DNO`, the expected block accesses is:

$$\frac{1}{10} \cdot 750 + \frac{1}{10}(750 \cdot 2) + \cdots + \frac{1}{10}(750 \cdot 10) = 4125.$$

- Next, we consider a clustering index on DNO. The indexing blocking factor is
$$ibfr = \text{floor}(4096/15) = 273.$$
We have 10 data blocks, so we need
$$ib = \text{ceil}(10/273) = 1$$
block to store all the index files. When executing the query, we find the block pointer corresponding to `DNO = x` in 1 block access. Then, we load the 750 contiguous blocks to find all the relevant tuples. This requires 751 block accesses.

Example 3.2.3. Assume we have the `EMPLOYEE` table with non-ordering key attribute `SSN`.

- We have $r = 300\ 000$ records.
- A record takes $R = 100$ bytes.
- The block size is $B = 4\ 096$ bytes.
- The attribute `SSN` occupies 9 bytes.
- A pointer occupies 6 bytes.

We want to run the query

```
SELECT   *
FROM     EMPLOYEE
WHERE    SSN = x;
```

Compute the number of block accesses we need using a linear scan and a secondary index on a non-ordering key attribute `SSN`.

The blocking factor is
$$bfr = \text{floor}(B/R) = 40$$
records per block. So, we need
$$b = \text{ceil}(r/bfr) = 7500$$
blocks to store all the data.

- First, we consider a linear search. Since the attribute `SSN` is key, this would take on average
$$b/2 = 3750$$
block accesses.

- Next, we consider using the secondary index. An index entry takes 15 bytes. So, the index blocking factor is

  $$ibfr = \text{floor}(4096/15) = 273.$$

  Since we are using a dense index, we have 300 000 index entries. So, we need

  $$ib = \text{ceil}(300\ 000/273) = 1099$$

  blocks.

  To search for a single employee, we first search within the index. Since the index file is sorted, we can find it in

  $$\log_2(ib) \approx 10.1$$

  block accesses. Next, we load the unique block pointed by the index entry. So, we require 11.1 block accesses to find the relevant tuple.

Example 3.2.4. Now, assume that we have the `EMPLOYEE` table on the non-ordering key attribute `SSN`.

- We have $r = 300\ 000$ records.
- A record takes $R = 100$ bytes.
- The block size is $B = 4\ 096$ bytes.
- The attribute `SSN` occupies 9 bytes.
- A pointer occupies 6 bytes.

What level of multilevel index should we build? Use this value to predict the number of block accesses when using the index to find a tuple given the `SSN` value.

The blocking factor is

$$bfr = \text{floor}(B/R) = 40$$

records per block. So, we need

$$b = \text{ceil}(r/bfr) = 7500$$

blocks to store all the data.
An index entry has tuples with attributes `SSN` and a pointer. So, this takes 15 bytes. In that case, the index blocking factor is

$$m = \text{floor}(4096/15) = 273.$$

Therefore, at L1 index, we have

$$b_1 = \text{ceil}(300\ 000/m) = 1099$$

index blocks, since the index is dense. So, at L2, we have 1 099 index entries, and we require

$$b_2 = \text{ceil}(b_1/m) = 5$$

blocks to store them. Furthermore, at level-3, we have 5 index entries, and we just require

$$b_3 = \text{ceil}(b_2/m) = 1$$

block for it. Since we just need 1 block here, we build a 3-level index. Now, to find the unique employee we are searching for, we just need 4 block accesses- 3 going from one index level to a lower level, and the last one to get the data block.

3.3 B and B+ Trees

Example 3.3.1. Assume that we have created a 3-level B tree of order $p = 23$ that is 69% full. Compute the number of data and tree pointers we can store in the tree. Moreover, assume the following.

- The block size is $B = 512$ bytes.
- A data pointer takes $Q = 7$ bytes.
- A tree pointer takes $P = 6$ bytes.
- A key value takes $V = 9$ bytes.

Using this information, compute the number of blocks we have in total.

In each block, we store

$$69\% \times 23 \approx 16$$

tree pointers, and 15 data pointers. In that case, the root block contains 16 tree pointers and 15 data pointers. At level 1, we have 16 blocks, each of which contains 16 tree pointers and 15 data pointers. At level 2, we have 16^2 blocks, each of which contains 16 tree pointers and 15 data pointers. At level 3, we have 16^3 blocks, each of which contains no tree pointers and 15 data pointers. This is summarised in the table below.

level	data pointers	tree pointers
0	15	16
1	$16 \cdot 15$	16^2
2	$16^2 \cdot 15$	16^3
3	$16^3 \cdot 15$	0

In total, we can store

$$15 + 16 \cdot 15 + 16^2 \cdot 15 + 16^3 \cdot 15 = 65\,535$$

data pointers and

$$16 + 16^2 + 16^3 = 4368$$

tree pointers.
A block has size

$$15 \cdot (7 + 9) + 16 \cdot 6 = 336 \text{ bytes.}$$

So, the blocking factor

$$bfr = \text{floor}(512/336) = 1.$$

In that case, we need

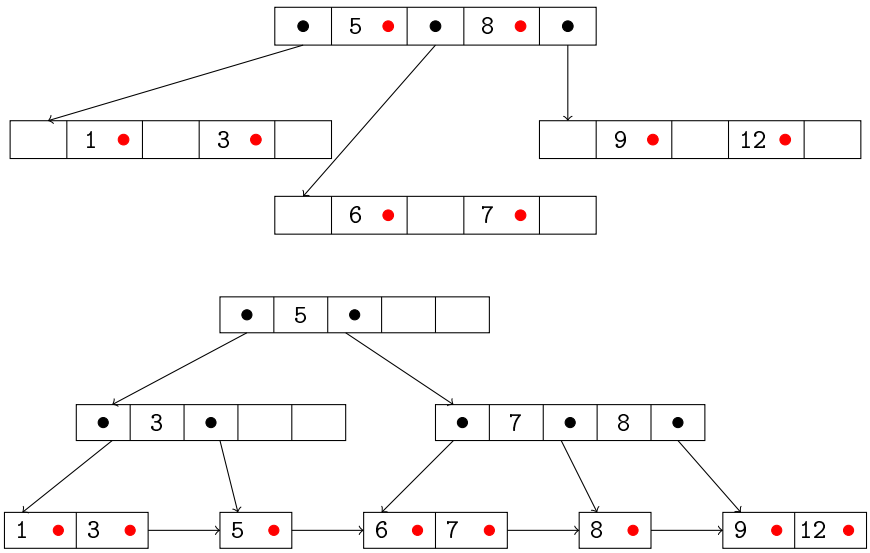
$$1 + 16 + 16^2 + 16^3 = 4369$$

blocks to store the 3-level B tree.

Example 3.3.2. Assume we have the following query.

```
SELECT   *
FROM     EMPLOYEE
WHERE    SSN >= 3 AND SSN <= 10;
```

We are also given the following B and B+ Trees.



Compute the number of block accesses we need in order to execute this query for both a B Tree and a B+ Tree.

- First, we use the B Tree. We access the root block and get the values `SSN = 5` and `SSN = 8`. Since they are both in the range, we need to access all the 3 branches.

  So, we need 4 block accesses within the B Tree, and 6 data block accesses.

- Next, we use the B+ Tree. We access the root block and get the value `SSN = 5`. We want to find the tuple satisfying `SSN = 3`, so we just need to take the left branch. Now, we take the left branch in level 1. Then, we have found the first block that we require. We can make use of the sorted structure of the leaf nodes to find all the other tuples that satisfy the given condition.

  So, we need 2 block accesses within the internal nodes of the B+ Tree, 5 block accesses within the leaf nodes of the B+ Tree, and 6 data block accesses.

Example 3.3.3. Construct a 3-level B+ Tree of order $p = 34$ and $p_L = 31$ that is 69% full. Also, compute the number of blocks we have in total.

We can fit

$$69\% \times 34 = 23$$

tree pointers in an internal node and

$$69\% \times 31 = 21$$

data pointers in a leaf node. The table below summarises the number of tree pointers and key values we can hold in the first 2 levels.

level	key values	tree pointers
0	22	23
1	$23 \cdot 22$	23^2
2	$23^2 \cdot 22$	23^3

So, we can fit 12 720 tree pointers. At level 3, we can fit $23^3 \cdot 21 = 255\,507$ key values and data pointers. Assuming that one B+ block fits one node, we require

$$1 + 23 + 23^2 + 23^3 = 12\,720$$

blocks.

Example 3.3.4. Assume that we are given the following query.

```
SELECT  *
FROM    EMPLOYEE
WHERE   SSN = k;
```

We are told the following:

- the size of a block $B = 512$ bytes;
- the number of tuples $r = 100\,000$;
- the blocking factor $bfr = 10$ records/block;
- a pointer takes up 10 bytes;
- the indexing key takes 10 bytes.

Moreover, we have the following primary access paths:

- A secondary index over `SSN`.
- A B+ tree of order 25 over `SSN`.

Compute the number of block accesses we require to execute the query using a linear search, secondary index and the B+ Tree.

- First, we consider a linear search. Since there are 100 000 records and $bfr = 10$, we have 10 000 data blocks. On average, the linear search will require 5000 block accesses since `SSN` is a key attribute.
- Next, we consider a secondary index. Since the index is secondary, we need a dense index- every data record appears in the index, with a data pointer (10 bytes) and the key value (10 bytes). So, each record takes up 20 bytes. We know a block can store 512

bytes, so each block can hold

$$\text{floor}(512/20) = 25$$

tuples. In that case, we require

$$\text{ceil}(100\ 000/25) = 4\ 000$$

extra blocks to store the secondary index. Moreover, using the secondary index, we require

$$\log_2 4000 \approx 12$$

block accesses to find the right tuple on the secondary index file, and then a further data block access to retrieve the tuple. In total, the search requires 13 block accesses.

- Finally, we consider a B+ tree index of order 25. Assuming the nodes are completely full, each internal node can hold 24 key values and 25 tree pointers, and a leaf block can hold 24 key values and data pointers. A pointer and an indexing key take 10 bytes, so an internal node takes up

$$25 \cdot 10 + 24 \cdot 10 = 490$$

bytes. Moreover, a leaf node takes up

$$24 \cdot 10 + 24 \cdot 10 + 10 = 490$$

bytes. So, we can fit 1 internal/leaf node per block. Next, we compute the level of B+ Tree we need to store the 100 000 tuples:

level	key values	tree pointers
0	24	25
1	$25 \cdot 24$	25^2
2	$25^2 \cdot 24$	25^3

So, at level 3, we have

$$25^3 \cdot 24 = 375\ 000$$

data pointers/key values. Since we only have 100 000 tuples, we are only using up about 30% of the leaf nodes. Moreover, we need

$$1 + 25 + 25^2 + 25^3 = 16\ 276$$

extra blocks to store the B+ tree.

Nonetheless, we can find a tuple using a B+ tree with just 5 block accesses:

1. we access the root internal node,
2. we descend to the relevant level 1 internal node,
3. we descend to the relevant level 2 internal node,
4. we descend to the relevant level 3 leaf node, and
5. we retrieve the relevant data block.

Example 3.3.5. Assume that we have the query

```
SELECT   AVG(SALARY)
FROM     EMPLOYEE
WHERE    SSN >= L AND SSN <= U;
```

Here,

- there are $b = 1\ 250$ blocks,
- there are $n = 1\ 250$ employees,
- the blocking factor $bfr = 1$ employee/block,
- `SSN` takes up 10 bytes,
- a pointer takes up $P = 10$ bytes,
- the B+ tree is of order $p = 5$ and $p_L = 10$,
- the leaf nodes are full,
- the SSN values range from 1 to 1250.

Find the maximum ratio

$$\alpha = \frac{U - L}{1250}$$

of `SSN` values selected by the query such that searching using the B+ Tree is more efficient than a linear scan.

We know that an internal node can hold 4 key values and 5 data pointers. So, we need a 3-level B+ tree so that the leaf nodes can hold the 1 250 records. The table below shows how many entries are present in the internal nodes.

level	key values	tree pointers
0	4	5
1	$5 \cdot 4$	5^2
2	$5^2 \cdot 4$	5^3

So, at level 3, we precisely have

$$5^3 \cdot 10 = 1\ 250$$

key values/data pointers.
When we execute this range query over the B+ tree, we need 3 block accesses to descend from the root to the leaf node containing the data pointer for `SSN = L`. Each leaf node contains 10 blocks, so we approximately need to access

$$\frac{U - L}{10} = \frac{1250\alpha}{10} = 125\alpha$$

leaf blocks. Moreover, we need to access all $U - L = 1250\alpha$ data blocks. Overall, we need to access

$$3 + 125\alpha + 1250\alpha = 3 + 1375\alpha$$

blocks.
Using a linear search, we need to access all the 1250 blocks since the tuples are not sorted with respect to `SSN`. Therefore, for B+ trees to be useful, we require

$$3 + 1375\alpha < 1250 \implies \alpha < \frac{1247}{1375} \approx 0.906.$$

So, if the ratio is less than 90.6%, then we should use B+ tree. Otherwise, we should use linear searching.

CHAPTER 3

QUERY OPTIMISATION

3.1 Query Processing

Almost all SQL queries involve sorting of tuples with respect to sorting requests defined by the user, e.g.

- `CREATE PRIMARY INDEX ON EMPLOYEE(SSN)` means that we sort by `SSN`
- `ORDER BY Name` means that we sort by `Name`
- `SELECT DISTINCT Salary` means that we sort by `Salary` to create clusters, and then identify the distinct values
- `SELECT DNO, COUNT(*) FROM EMPLOYEE GROUP BY DNO` means that we sort by `DNO` to create clusters, and then count the number of values per cluster.

Normally, we cannot store the entire relation into memory for sorting the records. So, to sort the tuples, we use external sorting algorithm.

External Sorting

The external sorting algorithm is a divide and conquer algorithm. We first divide a file of b blocks into L smaller sub-files (so each sub-file has b/L blocks). We require each of the sub-file to fit in memory. We load each small sub-file into memory and sort them (e.g. using quicksort) and then write it back to the disk. At the end of this, the sub-files are sorted.

We then merge sorted sub-files to generate bigger sub-files. We do this by loading the sub-file and combining them (using an algorithm similar to the mergesort merge algorithm). This process continues until we have combined it into the entire file. Note that we do not need to load the entire sub-file in one go to merge. We just load the blocks with similar values to sort a subsection of the sub-files.

The expected cost of external sorting is

$$2b(1 + \log_M L)$$

block accesses, where:

- b is the number of file blocks,
- M is the degree of merging (i.e. the number of sorted blocks merged in each loop), and
- L is the number of the initial sorted sub-files (before entering the merging phase).

This method is expensive as it is linear with respect to the number of blocks. Moreover, as the value of M increases, the number of block access decreases.

Executing queries

We will be considering queries of the form

```
SELECT   *
FROM     <relation>
WHERE    <selection-conditions>
```

Key query

First, assume that selection condition is just a key attribute. An example of such a query is

```
SELECT   *
FROM     EMPLOYEE
WHERE    SSN = "123"'
```

- If we use a linear search, then the expected cost is $b/2$ block accesses, where b is the number of blocks.
- Instead, we can also use a binary search. If the files are sorted with respect to the key attribute, the expected cost is $\log_2 b$ block accesses. Otherwise, we need to sort the files in $2b(1+\log_M L)$ block accesses, and then find the tuple in $\log_2 b$ block accesses.
- If we have a primary index of level t over the key, then it takes $t+1$ block accesses.
- Moreover, if we have a hash file, then it takes $1+O(n)$ block accesses, where n is the number of overflown buckets.
- Finally, if the file is not sorted over the attribute and we have a secondary index (B+ Tree of level t), then we require $t+1$ block accesses.

Range query

Next, assume that the selection condition is a range query with respect to a key attribute. An example of such a query is

```
SELECT   *
FROM     DEPARTMENT
WHERE    DNumber >= 5;
```

If we have a primary index of level t over the key, then it takes $t+O(b)$ block accesses, where b is the number of blocks in the worst case scenario. Since a primary index relation is sorted, we first find the value `DNumber = 5` and then keep going lower.

Non-key query

Now, assume that the selection condition is with respect to an ordering, non-key field. An example of such a query is

```
SELECT   *
FROM     EMPLOYEE
WHERE    DNO = 5;
```

- If the file is sorted with respect to the attribute and we have a clustering index is of level t, then the expected cost is $t + O(b/n)$, where b is the number of blocks and n is the number of distinct values of the attribute. This is under the assumption that the attribute is uniformly distributed over the tuples.
- If the file is not sorted with respect to the attribute and we have a secondary index (B+ Tree of level t), then we need $t+1+O(b)$ block accesses, where b is the number of block pointers. Here, the B+ Leaf nodes point to a block of pointers which further point to data blocks.

Disjunctive query

Now, assume we have a disjunctive select statement, i.e. we have an `OR` present in the `WHERE` clause. An example is given below.

```
SELECT *
FROM EMPLOYEE
WHERE SALARY > 10000 OR NAME LIKE '%Chris%';
```

The result will be the union of tuples satisfying the two conditions.

If an access path exists (e.g. B+, hash, primary index) index for all the attributes, we use each of them to retrieve the set of records satisfying each condition. Then, we return the union of the sets to get the final result. Otherwise, we must use a linear search.

Conjunctive query

On the other hand, assume that we have an conjunctive select statement, i.e. we have an `AND` present in the `WHERE` clause. An example is given below.

```
SELECT *
FROM EMPLOYEE
WHERE SALARY > 10000 AND NAME LIKE '%Chris%';
```

The result will be the intersection of tuples satisfying the two conditions.

If an access path exists for any of the attribute, we can use that to construct an intermediate result. Then, we can use a linear search to validate the other conditions. If none of the attributes have an attribute path, we need to use linear search. If we have multiple access paths, we should choose the right index to generate the smallest intermediate result. We predict the selectivity (the number of tuples received) for each attribute to do this. This is query optimisation.

Join queries

Next, assume that we have a join query. Joining is the most resource-consuming operator. We will only be considering two-way equijoin. These queries are of the following form.

```
SELECT  *
FROM    R, S
WHERE   R.A = S.B;
```

There are 5 ways for processing join queries- naive join, nested-loop join, index-based nested-loop join, merge-join and hash-join.

Naive join

In naive join, we compute the Cartesian product of `R` and `S`, and then filter the ones that satisfy `R.A = S.B`. This method is inefficient because we get rid of most of the tuples from the Cartesian product most of the time.

Nested loop join

In nested-loop join, we have a nested loop algorithm over the two relations to only generate products if they satisfy the join condition. In terms of relations, the algorithm is the following.

```
for each tuple r in R:
    for each tuple s in S:
        if r.A = s.B:
            add(r, s) to the result file;
```

We say `R` is the outer relation and `S` is the inner relation. The choice of outer and inner relation affects the computation process- it is more efficient to have the file with fewer blocks in the outer loop. Since we only have access to blocks, the algorithm is the following in terms of blocks.

- Load a set of blocks from the outer relation `R`.
- Load one block from inner relation `S`.
- Maintain an output buffer for the matching tuples (r, s) using the algorithm above.
- Join the `S` block with each `R` block from the chunk.
- For each matching tuple r in `R` and s in `S`, add (r, s) to the output buffer.
- If the output buffer is full, pause and write the current join result to the disk.
- Load the next `S` block.
- After going through all the `R` blocks, go to the next set of `R` blocks.

Index-based nested-loop join

If we have an index on either `A` or `B`, we can make use of it in the join. Assume that we have an index I on `S.B`. Then, the algorithm becomes:

```
for each tuple r in R:
    use index of B to retrieve all tuples s in S
        satisfying s.B = r.A
    for each such tuple s, add (r, s) to the result file;
```

This process is much faster than nested-loop join. If we have two indices, we need to choose the right one to minimise the join processing cost.

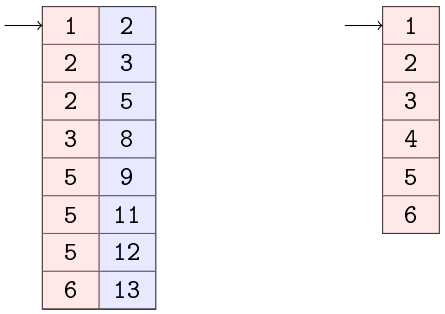
Merge-join

Now, assume that both `R` and `S` are physically ordered on their joining attribute `A` and `B`. Then, we can use the following approach to join them:

- Load a pair (R', S') of sorted blocks into memory.
- Scan the two blocks concurrently over the joining attribute (sort-merge algorithm).
- If matching tuples are found, then store them in a buffer.

In this case, the blocks of each file are scanned only once. But, if `R` and `S` are not physically sorted, then we need to use the external sorting method.

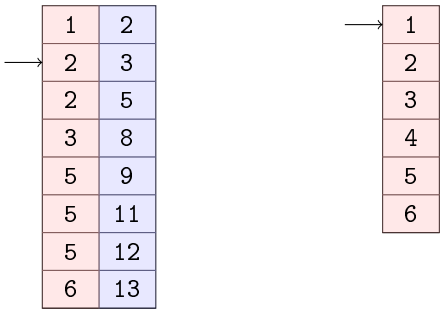
For example, assume that we have the following blocks.



1	2
2	3
2	5
3	8
5	9
5	11
5	12
6	13

1
2
3
4
5
6

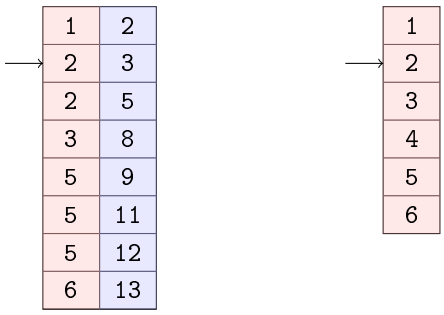
Since the join query is based on a foreign key and a primary key, one of these blocks must have unique entries. In this case, it is the block on the left. We always go down the non-unique set of entries more frequently (if present) than the unique set of entries. At this point, there is a match between the two values in the blocks. We move down the non-unique block.



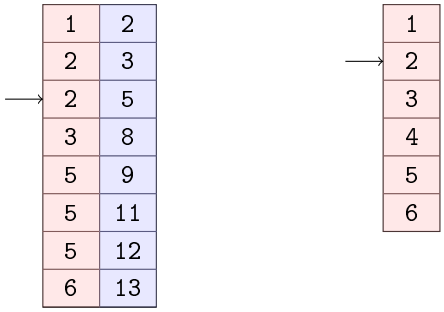
1	2
2	3
2	5
3	8
5	9
5	11
5	12
6	13

1
2
3
4
5
6

There is a mismatch here- the entry `2` on the left is bigger than the entry `1`. So, we move down the right block as well.



There is a match in this case as well. We move down the non-unique block again.



There is another match in this case. We continue this process to join the two blocks.

Hash join

Now, if `R` and `S` are partioning into M buckets with the same hash function over the join attributes `A` and `B`, then we can use a hash-join algorithm. Assuming `R` is the smallest file and fits into main memory, i.e. M buckets of `R` are in memory, we start by partioning.

```
for each tuple r in R:
    compute y = h(r.A) // the address of the bucket
    place tuple r into bucket y = h(r.A) in memory
```

Next, we have the probing phase.

```
for each tuple s in S:
    compute y = h(s.B) // using the same hash function
    find the bucket y = h(s.B) in memory
    for each tuple r in R in the bucket y:
        if s.B = r.A:
            add(r, s) to the result file;
```

In this case, we need to access each block precisely once. If `R` does not fit in memory, then we need $3(b_R + b_S)$ block accesses.

3.2 Selection selectivity

There are two fundamental concepts in optimisation- join and selection selectivity. Selection selectivity is the fraction of tuples satisfying a condition. Join selectivity is the fraction of matching tuples in the Cartesian product.

Before running a query, we would like to predict the selection and the join selectivity, and hence the number of blocks we expect to retrieve. Then, using the actual block accesses, we aim to refine the expected cost. The expected cost is expressed as a function of selectivity. Using this result, we choose the optimal strategy to run a query.

In query optimisation, we are given a query as an input. The output is the optimal execution plan. There are two types of query optimisations:

- Heuristic optimisation- we transform a SQL query into an equivalent and efficient query using Relational Algebra.
- Cost-based optimisation- we provide many execution plans and estimate their costs, and choose the plan with the minimum cost.

The cost function is what we want to optimise. It has parameters- the number of block accesses, the memory requirements, the CPU computational cost, the network bandwith, etc. We will consider the number of block accesses and memory requirements.

We will use statistical information available to estimate the execution of a query. To do this, we store the following information for each relation:

- the number of records r, and the (average) size of each record R.
- the number of blocks b, and the blocking factor f.
- the primary file organisation (heap, hash or sequential).
- the available indices- primary, clustering, secondary or B+ Trees.

For each attribute, we further store the following:

- the number of distinct values (NDV) n of the attribute.
- the domain range- the minimum and the maximum value of the attribute (among the tuples).
- the type of attribute- continuous or discrete, key or non-key.
- level t of Index of the attribute, if present.
- the probability distribution function $P(A = x)$, which indicates the frequency of each value x of the attribute A in the relation.

A good approximation of the distribution is a histogram.

Selection selectivity of an attribute A, denoted by $sl(A)$, is the fraction of tuples that satisfy a given condition. Therefore, its value is between 0 and 1. If $sl(A) = 0$, then none of the records satisfy this condition with respect to the attribute A. On the other hand, if $sl(A) = 1$, then all the records satisfy the condition with respect to the attribute A. As a probability, the selection

selectivity tells us how likely is it for a tuple to satisfy the given condition with respect to an attribute.

The selection cardinality s is the number of tuples we expect to satisfy a query with respect to a given attribute. In other words, the selection cardinality

$$s = sl(A) \cdot r,$$

where r is the number of tuples present. So, the value is between 0 and r. We predict this value without scanning any block. For example, if $r = 1000$ and $sl(A) = 0.3$, then the selection cardinality $s = 300$.

To predict selectivity, we can approximate the distribution of attribute values using a histogram. This allows us to maintain an accurate selectivity estimate. However, we must maintain this histogram whenever there is a change to the attribute for any of the tuples. In this case, we assume that

$$sl(A = x) \approx P(A = x).$$

That is, the selection selectivity is approximately equal to the probability that the value of the attribute is that value.

Another possible approximation is that all values are uniformly distributed. This means that we do not need to maintain a histogram- all the values are of equal probability, so we barely need to store one value. However, we provide a less accurate prediction for the selection selectivity. In this case, we assume that

$$sl(A = x) \approx \frac{1}{n},$$

where n is the number of distinct values of A. Note that the selectivity is a constant value independent of x.

If A is a key attribute, then

$$sl(A = x) \approx \frac{1}{r}$$

is a good estimate. There is only one tuple that satisfies the condition, so the selection cardinality $s = 1$. So, the uniformity assumption is valid in this case. We do not need to build a histogram.

Now, if A is a non-key attribute, with $n = NDV(A)$, the number of distinct values of A present. Then, the estimate

$$sl(A = x) \approx \frac{1}{n}$$

is not a good estimate. This is under the assumption that all the records are uniformly distributed across the n distinct values. This is not true in general.

Next, consider the following range selection selectivity.

```
SELECT   *
FROM     <relation>
WHERE    A >= x;
```

The domain range is defined to be $\max(A) - \min(A)$, and the query range in this case is $\max(A) - x$. If $x > \max(A)$, then $sl(A \geq x) = 0$. Otherwise,

$$sl(A \geq x) = \frac{\max(A) - x}{\max(A) - \min(A)},$$

under the uniformity assumption.

In particular, if we have $r = 1000$ employees, the attribute value ranges from 100 to 10 000 and $x = 1000$, then the selection selectivity is

$$sl(A \geq 1000) = \frac{\max(A) - x}{\max(A) - \min(A)} = \frac{10\ 000 - 1000}{10\ 000 - 100} = 0.909$$

In that case, the selection cardinality is 909 tuples.

Now, consider the conjunctive selectivity, i.e. a query satisfying 2 conditions. An example is given below.

```
SELECT  *
FROM    <relation>
WHERE   (A = x) AND (B = y);
```

If the two attributes A and B are statistically independent, then the selectivity of the conjunction query q is

$$sl(q) = sl(A = x) \cdot sl(B = y).$$

Now, we consider a disjunctive selectivity condition, i.e. a query satisfying one of 2 conditions. An example is given below.

```
SELECT  *
FROM    <relation>
WHERE   (A = x) OR (B = y);
```

Then, the selection selectivity of this query is

$$sl(Q) = sl(A) + sl(B) - sl(A)sl(B).$$

Here as well, we assume that the two attributes A and B are statistically independent.

Now, assume that we have r tuples, and the attribute A has n number of distinct values. Under the uniform assumption of the attribute, the expected number of block accesses is

$$\text{ceil}(s/f) = \text{ceil}(r/f \cdot n),$$

where s is the selection cardinality and f is the blocking factor. The graph below summarises this.

Now, we apply this formula to refine the selection cost. Assume that we have the query

```
SELECT  *
FROM    <relation>
WHERE   A = x;
```

We have b blocks; the blocking factor is f; we have r records; and the number of distinct values of the attribute A is n. We propose different approaches and the expected number of block accesses in each case.

- Assuming that the tuples are sorted with respect to the attribute A, we can use a binary search algorithm.
  - If A is a key attribute, then the expected cost is $\log_2 b$ block accesses. It is independent of the selection selectivity.

  - If A is not a key attibute, then it takes $\log_2 b$ block accesses to find the first block with record $A = x$. We then access all the contiguous blocks whose records satisfy $A = x$. This takes $\text{ceil}(s/f) - 1$ further block accesses. In total, the expected cost is

$$\log_2 b + \text{ceil}(s/f) - 1 = \log_2 + \text{ceil}(r \cdot sl(A)/f) - 1.$$

    This is a function of the selection selectivity.

- Now, assume that we have a level-t multilevel primary/clustering index on the attribute.
  - If A is a key attribute, then the expected cost is $t+1$ block accesses. It is independent of the selection selectivity.
  - If A is a non-key attribute, then it takes t block accesses to descend the tree. The selection cardinality is $s = r \cdot sl(A)$ tuples. So, we require $\text{ceil}(s/f)$ block accesses, where f is the blocking factor. So, the expected cost is

$$t + \text{ceil}(s/f) = t + \text{ceil}(r \cdot sl(A)/f).$$

    This is a function of the selection selectivity.

- Assume next that we have a level-t B+ tree on the attribute.
  - If A is a key attribute, then the expected cost is $t+1$ block accesses. It is independent of the selection selectivity.
  - If A is a non-key attribute, then it takes t block accesses to descend the tree. At this point, we have access to the block pointer to all the blocks containing a tuple satisfying the condition. The selection cardinality is $s = r \cdot sl(A)$ tuples. Since the tuples are not sorted with respected to A, we assume that we must access s blocks to retrieve all the tuples. In that case, the expected number of block accesses is

$$t + 1 + s = t + 1 + r \cdot sl(A).$$

    This is a function of the selection selectivity.

- Finally, assume that we are using a hash file structure. For any attribute A, then the expected cost is just 1 block access (if there are no overflown buckets). This is independent of the selection selectivity.

Next, assume that we have the query

```
SELECT   *
FROM     <relation>
WHERE    A >= x;
```

- Assume further that we have a level t primary index and A is a key attribute. We traverse the tree in t block access, finding the data block for $A = x$. Then, the range selection cardinality is

$$s = r \cdot sl(A \geq x).$$

So, we further access ceil(s/f) blocks since the file is sorted with respect to this attribute. In total, we have

$$t + \text{ceil}(s/f) = t + \text{ceil}(r \cdot sl(A)/f)$$

block accesses. This is a function of the selection selectivity.

- Now, assume that we have a level t B+ tree and A is a key, non-ordering attribute. We traverse the tree in t block access, finding the data block for $A = x$. Then, the range selection cardinality is

$$s = r \cdot sl(A \geq x).$$

Since the file is not sorted with respect to this attribute, we further access s blocks in the worst case. In total, we have

$$t + s = t + r \cdot sl(A)$$

block accesses.

3.3 Join selectivity

We measure join selectivity with respect to the cartesian product of the two relations. For example, consider the following tables.

SSN	Name	DNO
1	Chris	D1
2	Stella	D1
3	Phil	D2
4	Thalia	D3
5	John	D3

DNumber	MGR_SSN	DName
D1	1	HR
D2	3	R&D
D3	4	SPR

Their Cartesian product has $5 \cdot 3 = 15$ entries. However, the following is the joined table with respect to `DNO` and `DNumber`.

SSN	Name	DNO	MGR_SSN	DName
1	Chris	D1	1	HR
2	Stella	D1	1	HR
3	Phil	D2	3	R&D
4	Thalia	D3	4	SPR
5	John	D3	4	SPR

There are only 5 entries here, so the join cardinality is 5. Moreover, the join selectivity is $5/15 = 1/3$.

If we are joining two relations R and S, then the join selectivity is given by

$$js = |R \bowtie S|/|R \times S|.$$

It is a value between 0 and 1. Moreover, the join cardinality is given by

$$jc = js \cdot |R||S|.$$

We can predict the join selectivity/cardinality using the join selectivity theorem. Given $n = \text{NDV}(A, R)$ and $m = \text{NDV}(B, S)$, then

$$js = 1/\max(n, m).$$

This implies that

$$jc = |R||S|/\max(n, m).$$

We illustrate the this with an example. Assume that we have a relation `E` of employees and `D` of dependents. An employee can have multiple dependents. Their common attribute is `SSN` of the employee. We have

- $n = \text{NDV}(\text{SSN}, \texttt{E}) = 2000$ and $|E| = 2000$;
- $m = \text{NDV}(\text{SSN}, \texttt{P}) = 3$ and $|P| = 5$.

Therefore, the join selectivity is

$$1/\max(n, m) = 1/2000 = 0.0005.$$

This implies that there is a 0.05% probability of getting a matching tuple from the cartesian space.

Now, assume we have relations R and S with b_R and b_S blocks. They can be joined using `R.A = S.B`. In memory, we have n_B blocks, and there are n distinct values of `R.A` and m distinct values of `S.B`. The blocking factor is f_{RS} matching tuples per block. The size of the maching tuple (r, s) is the sum of the size of the tuple r and s. We wrtie every full result block to disk. We expect the result to have

$$jc = js \cdot |R||S| = |R||S|/\max(n, m).$$

So, the number of result blocks is

$$k = (js \cdot |R| \cdot |S|)/f_{RS}.$$

When executing a join query, this is the number of block accesses corresponding to store the result back into memory. This can be used for further processing of the data when it doesn't fit in memory, for example.

CHAPTER 4

EXAMPLES- CHAPTER 4

4.1 Query Processing

Example 4.1.1. Consider the following query.

```
SELECT  *
FROM    EMPLOYEE E, DEPARTMENT D
WHERE   E.DNO = D.DNUMBER;
```

Assume we have n_E Employee blocks and n_D Department blocks. Moreover, the memory can store n_B blocks. Predict the cost of using a nested-loop join algorithm to execute the query, with `EMPLOYEE` in the outer loop and `DEPARTMENT` in the inner loop.

In memory, we require a block for reading the inner file `D` and a further block to write the join result. The other $n_B - 2$ blocks can be used to read the outer file `E`. This is the chunk size.
When we run the nested-loop algorithm, we load $n_B - 2$ blocks for the outer relation, and then load each block from the inner relation one by one and check whether there are any possible tuples we can output. So, the outer loop runs for

$$\frac{n_E}{n_B - 2},$$

and the inner loop is n_D. The total number of block accesses is therefore

$$n_E + \frac{n_E}{n_B - 2} \cdot n_D.$$

Example 4.1.2. Assume we have the following query.

```
SELECT  *
FROM    EMPLOYEE E, DEPARTMENT D
WHERE   D.MGR_SSN = E.SSN;
```

We have the following primary access paths:

- A B+ Tree on `Mgr_SSN` with level $x_D = 2$.
- A B+ Tree on `SSN` with level $x_E = 4$.

Moreover, the record `EMPLOYEE` has $r_E = 6\,000$ tuples in $n_E = 2\,000$ blocks, and the record `DEPARTMENT` has $r_D = 50$ tuples in $n_D = 10$ blocks. Compute the number of expected block accesses using an index-based nested-loop join algorithm.

- If we use index-based nested-loop on the B+ Tree for Department relation, we are searching the B+ tree to check whether a given `E.SSN` is also `D.Mgr_SSN`. However, since not every employee is

a manager, many of these searches fail. The probability of an employee being a manager is

$$50/6\ 000 = 0.83\%.$$

So, 99.16% of searching using the B+ Tree is meaningless. During the process, we load each employee block, and for each tuple, we traverse the B+ Tree to find whether the SSN corresponds to a manager. This requires

$$n_E + r_E \cdot (x_D + 1) = 2\ 000 + 6\ 000 \cdot (2 + 1) = 20\ 000,$$

block accesses.

- Instead, if we use the B+ tree for Employee relation, then we require
$$n_D + r_D \cdot (x_E + 1) = 10 + 50 \cdot 5 = 260$$
block accesses. The probability of a manager being an employee is 100%, so this approach is much more efficient.

Example 4.1.3. Assume we have the following query.

```
SELECT   *
FROM     EMPLOYEE E, DEPARTMENT D
WHERE    D.MGR_SSN = E.SSN;
```

The record `EMPLOYEE` has $r_E = 6\ 000$ tuples in $n_E = 2\ 000$ blocks, and the record `DEPARTMENT` has $r_D = 50$ tuples in $n_D = 10$ blocks. Compute the number of block accesses we need if we use the sort-merge-join algorithm when the two relations are sorted and when they are not sorted with respect to the joining attribute.

- If the two files are sorted, we load each block from the two relations precisely once, so this requires
$$n_E + n_D = 2010$$
block accesses.
- If both files are not sorted, we need to use external sorting algorithm to sort them. The external sorting process for the `Employee` relation requires
$$2n_E + 2n_E \log_2(n_E/n_B)$$
block accesses- the value n_E/n_B is the number of sub-files initially sorted, where n_B is the number of available blocks in memory. We might have to sort the `Department` relation requires
$$2n_D + 2n_D \log_2(n_D/n_B)$$
block accesses. In total, the sort-merge-join algorithm requires
$(n_E+n_D)+(2n_E+2n_E \log_2(n_E/n_B))+(2n_D+2n_D \log)2(n_D/n_B)) = 38\ 690$
block accesses if both the relations are not sorted.

Example 4.1.4. Assume we have the query

```
SELECT  *
FROM    EMPLOYEE E, DEPARTMENT D
WHERE   E.SSN = D.MGR_SSN;
```

We have $n_E = 100$ Employee blocks, $r_D = 100$ Department tuples in $n_D = 10$ blocks. In memory, we can store $n_B = 12$ blocks, and we have a 2-level index on `E.SSN`. Find the number of block accesses required using a nested-index join algorithm.

The algorithm runs as follows:

- For each department d:
  - Use the B+ Tree on `E.SSN` to find the Employee `d.MGR_SSN`.
  - Output the tuple `(d, e)`.

  So, we require

$$n_D + r_D(2+1) = 10 + 100 \cdot 3 = 310$$

  block accesses.

Example 4.1.5. Next, assume we have the query

```
SELECT  *
FROM    EMPLOYEE E, DEPARTMENT D
WHERE   E.SSN = D.MGR_SSN;
```

We have $n_E = 100$ Employee blocks, $r_D = 100$ Department tuples in $n_D = 10$ blocks. In memory, we can store $n_B = 12$ blocks, and we have a hash function on `D.MGR_SSN` that maps to 10 buckets. Find the number of block accesses required using a hash-join algorithm.

The algorithm runs as follows.

- Load all the Departments `d`. This fits in memory.
- For each Employee `e`,
  - Use the hash function to find the bucket for Department `d`.
  - Find the tuple `d` in the bucket.
  - Output the tuple `(d, e)`.

  So, we require

$$n_D + n_E = 10 + 100 = 110$$

  block accesses.

Example 4.1.6. Assume that we have the query

```
SELECT  *
FROM    EMPLOYEE E, EMPLOYEE S
WHERE   E.SUPER_SSN = S.SSN:
```

Assume we have $r_E = 10\ 000$ tuples in $n_E = 2\ 000$ blocks. Moreoer, assume that 10% of the employees are supervisors. We have a 5-level B+ Tree on `SSN` and 2-level B+ Tree on `Super_SSN`. Compute the number of expected block accesses using an index-based nested-loop join algorithm for both the B+ trees.

The index-based nested-loop join algorithm for either B+ Tree is the following.

- For each Employee `e`
- If `e.SUPER_SSN` is not null (i.e. the employee is not a supervisor)
  - use the B+ Tree to find Employee `s.SSN`
  - Output the tuple `(e, s)`.

Since 10% of the employees are supervisors, if we use a level-t B+ Tree, then we need to retrieve all the employee blocks, and then $t + 1$ block accesses for 90% of the non-supervisor employees. So, if we use the 5-level B+ Tree on `SSN`, then we require

$$n_E + 0.9 \cdot r_E(5 + 1) = 2000 + 0.9 \cdot 10000 \cdot (5 + 1) = 56000$$

block accesses. Instead, if we use the 2-level B+ Tree on `SUPER_SSN`, then we require

$$n_E + 0.9 \cdot r_E(2 + 1) = 2000 + 0.9 \cdot 10000 \cdot (2 + 1) = 29000$$

block accesses. The B+ Tree on `SUPER_SSN` requires fewer block accesses since it only indexes supervisors, while the B+ Tree on `SSN` indexes both supervisors and non-supervisors.

4.2 Selection selectivity

Example 4.2.1. Consider the following query.

```
SELECT  *
FROM    EMPLOYEE
WHERE   DNO = 5 AND SALARY = 40 000;
```

Assume that:

- the number of distinct values of salary NDV(`Salary`) = 100;
- the number of distinct values of department numbers NDV(`DNO`) = 10;
- there are $r = 1000$ employees that are evenly distributed among salaries and departments.

Compute the selection selectivity and the selection cardinality of the query.

The selection selectivity for the `Salary` attribute is

$$sl(\texttt{Salary}) = \frac{1}{100} = 0.01$$

Moreover, the selection selectivity for the DNO attribute is

$$sl(\texttt{DNO}) = \frac{1}{10} = 0.1.$$

So, under the assumption that salary is independent of the department, we find that the selection selectivity of the query Q is

$$sl(Q) = sl(\texttt{Salary}) \cdot sl(\texttt{DNO}) = 0.001.$$

In terms of the selection cardinality, this is 1 tuple.

Example 4.2.2. Consider the following query

```
SELECT  *
FROM    EMPLOYEE
WHERE   DNO = 5 OR SALARY = 40 000;
```

Assume that:

- the number of distinct values of salary NDV(`Salary`) = 100;
- the number of distinct values of department numbers NDV(`DNO`) = 10;
- there are $r = 1000$ employees that are evenly distributed among salaries and departments.

Compute the selection selectivity and the selection cardinality of the query.

The selection selectivity for the `Salary` attribute is

$$sl(\texttt{Salary}) = \frac{1}{100} = 0.01$$

Moreover, the selection selectivity for the DNO attribute is

$$sl(\texttt{DNO}) = \frac{1}{10} = 0.1.$$

So, under the assumption that salary is independent of the department, we find that the selection selectivity of the query Q is

$$sl(Q) = sl(\texttt{Salary}) + sl(\texttt{DNO}) - sl(\texttt{Salary}) \cdot sl(\texttt{DNO}) = 0.109.$$

In terms of the selection cardinality, this is 109 tuples.

Example 4.2.3. We are given the following query.

```
SELECT  *
FROM    EMPLOYEE
WHERE   DNO = 5 AND SALARY = 30000 AND EXP = 0;
```

Assume that:

- there are $r = 10\ 000$ records in $b = 2000$ blocks.
- the file is sorted with respect to the `Salary` attribute.
- there are 500 distinct salary values, 125 departments, and 2 values of `EXP` (experienced or inexperienced).
- we can fit 100 blocks in memory.

Moreover, we have built the following access paths:

- A clustering index on `Salary`, with 3 levels.
- A B+ Tree on `DNO`, with 2 levels.
- A B+ Tree on `EXP`, with 2 levels.

Find the best way to execute this query.

- We can linearly search all the tuples and return those tuples satisfying the query. This requires 2000 block accesses since none of the searching attributes are key.
- We can make use of the B+ Tree on `DNO`. Here, we first get all the tuples that satisfy `DNO = 5`. For this, we require 2 block accesses to descend the tree, 1 to get the block of pointers, and $s_{\texttt{DNO}}$ blocks to get the actual data blocks (in the worst case). Under the uniformity assumption, we find that the selection cardinality of `DNO` is
$$s_{\texttt{DNO}} = \frac{1}{125} \cdot 10\ 000 = 80.$$
Since we expect 80 tuples to satisfy this condition, we can fit all of them in 16 blocks (as $bfr = 5$). This fits in memory. So, after finding these tuples, we can linearly check whether they satisfy the other two conditions. Overall, we just require 83 block accesses.

- We can make use of the clustering index on `Salary`. For this, we require 3 block accesses to descend the multilevel index and get to the first level of index. Then, it takes

$$\text{ceil}(s_{\texttt{Salary}}/f)$$

block accesses to retrieve all the tuples satisfying `Salary = 30 000`, where $f = 5$ is the blocking factor. Under the uniformity assumption, we find that the selection cardinality of `Salary` is

$$s_{\texttt{Salary}} = \frac{1}{500} \cdot 10\ 000 = 20.$$

In that case,

$$\text{ceil}(s_{\texttt{Salary}}/f) = 4.$$

Since we expect 20 tuples to satisfy this condition, we can fit them in 4 blocks- this fits in memory. After finding these tuples, we can linearly check whether they satisfy the other two conditions. Overall, we just require 7 block accesses.

- We can make use of the B+ Tree on `EXP`. For this, we require 2 block accesses to descend the tree, 1 to get the block of pointers, and $s_{\texttt{EXP}}$ blocks to get the actual data blocks (in the worst case). Under the uniformity assumption, we find that the selection cardinality of `EXP` is

$$s_{\texttt{EXP}} = \frac{1}{2} \cdot 10\ 000 = 5000.$$

The 5000 tuples occupy 1000 blocks, so they cannot all fit in memory. One approach would be to write 900 of the blocks back and keep them for processing later. We would further need to read them at a later point to check that they satisfy the other conditions. This would require 6803 block accesses.

Clearly, the best way to execute this query is by using the clustering index on `Salary`, requiring 7 block accesses.

Example 4.2.4. We are given the following query.

```
SELECT  *
FROM    EMPLOYEE
WHERE   DNO = 5 OR (SALARY >= 500 AND EXP = 0);
```

Assume that:

- there are $r = 10\ 000$ records in $b = 2000$ blocks.
- the file is sorted with respect to the `Salary` attribute.
- there are 500 distinct salary values, 125 departments, and 2 values of `EXP` (experienced or inexperienced).
- we can fit 1100 blocks in memory.

- the minimum salary value is 100 and the maximum is 10 000.

Moreover, we have built the following access paths:

- A clustering index on Salary, with 3 levels.
- A B+ Tree on `DNO`, with 2 levels.
- A B+ Tree on `EXP`, with 2 levels.

Find the best way to execute this query.

- We can linearly search all the tuples and return those tuples satisfies the query. This requires 2 000 block accesses.
- We can make use of the access paths. First, we get all the tuples that satisfy `DNO = 5` using the B+ tree on `DNO`. For this, we require 2 block accesses to descend the tree, 1 to get the block of pointers, and $s_{\texttt{DNO}}$ blocks to get the actual data blocks (in the worst case). Under the uniformity assumption, we find that the selection cardinality of `DNO` is

  $$s_{\texttt{DNO}} = \frac{1}{125} \cdot 10\ 000 = 80.$$

  Since we expect 80 tuples to satisfy this condition, we can fit all of them in 16 blocks (as $bfr = 5$). This fits in memory. So, after finding these tuples, we can linearly check whether they satisfy the other two conditions. Overall, this require 83 block accesses.

  Next, we consider how we find all the tuples that satisfy `SALARY >= 500 AND EXP = 0`.

  - We can use the B+ Tree on `EXP`. Under the uniformity assumption, we find that the selection cardinality of `EXP` is

    $$s_{\texttt{EXP}} = \frac{1}{2} \cdot 10\ 000 = 5000.$$

    Since the tree has 2 levels, it takes 5003 block accesses to find the 5000 tuples that satisfy the condition. This fits in 1000 blocks, so we can store it in memory. Overall, this takes 5086 block accesses.
  - We can also use the B+ Tree on `Salary`. Under the uniformity assumption, we find that the selection cardinality of `Salary` is

    $$s_{\texttt{Salary}} = \frac{1}{500} \cdot 1000 = 20.$$

    It takes 1921 block access to find the 1918 blocks that satisfy the condition. This does not fit in memory. We can hold at most 1084 blocks in memory and then write the other 834 blocks back to the disc. After filtering blocks in the main memory, we free up some space in the memory. In the first iteration, we discard the blocks that do not satisfy `EXP = 0`.

Since there are 2 distinct values of `EXP`, we discard half of them. This gives us 542 free blocks. Now, we load 542 blocks from the disc and filter them in the same way. At this point, we have 292 blocks left to be filtered. Next, we get 271 free blocks. During this iteration, we have 21 further blocks to be filtered. In the next iteration, we can fully filter the blocks. In total, we require

$$1921 + 834 + 542 + 271 + 21 = 3589$$

block accesses.

Therefore, the best choice here is linear searching.

4.3 Join selectivity

Example 4.3.1. Assume we have the following query.

```
SELECT   *
FROM     EMPLOYEE E, DEPARTMENT D
WHERE    E.DNO = D.DNUMBER;
```

We have the following context:

- there are $r_E = 10\ 000$ employee tuples that fit in $b_E = 2000$ blocks;
- there are $r_D = 125$ department tuples that fit in $b_D = 13$ blocks;
- the blocking factor of the joined tuple is $f_{RS} = 4$;
- we can fit $n_B = 10$ blocks in memory.

Moreover, we have built the following primary access paths:

- Primary Index on DNUMBER $x_{Dnumber} = 1$ level.
- B+ Tree Index on DNO $x_{Dno} = 2$ levels.

Using this information, find the best approach to execute this query.

The selection selectivity of the `DNO` attribute in the `DEPARTMENT` relation is given by

$$sl(\texttt{DNO}) = 1/125 = 0.008,$$

under the uniformity assumption. An employee works in a department- this is the joining condition. Assuming that the department an employee works in is unique, the join selectivity is

$$js = 1/\max(125, 125) = 0.008.$$

This means that the join cardinality is

$$jc = js \cdot r_E \cdot r_D = 10\ 000$$

tuples. These fit in 2500 blocks.

- First, we consider a nested-loop join algorithm. If the department is in the outer loop, the following is the algorithm:
  - load $n_B - 2$ `DEPARTMENT` blocks.
  - load all the `EMPLOYEE` blocks, one at a time.
  - check whether any tuple in the `EMPLOYEE` block can be joined with a tuple in one of the `DEPARTMENT` blocks.
  - Restart until all the `DEPARTMENT` blocks have been loaded.

  Here, we are only storing one block during the execution to store the output (i.e. if the block is full, we write it back to memory). This is why we can use $n_B - 1$ blocks in memory to store the

`EMPLOYEE` and the `DEPARTMENT` blocks. So, the total number of blocks that we read is

$$b_D + (\text{ceil}(b_D/n_B - 2)) \cdot b_E = 4013.$$

We know that the join cardinality is 10 000 tuples, which fit in 2500 blocks. So, the total cost of executing this query (read and write is)

$$4013 + 2500 = 6513$$

block accesses.

- Next, we consider an index-based nested-loop join algorithm with employee as the outer relation. The following is the algorithm:
  - Load an `EMPLOYEE` block.
  - Use the primary index on `DNUMBER` to find the relevant block (and then the tuple) to join.
  - Output the result.
  - Continue for all the `EMPLOYEE` blocks.

  So, the total number of blocks that we read is

  $$b_E + r_E(x_{\texttt{DNumber}+1}) = 22\ 000.$$

  If we write the result back into the disc, this requires 2500 further block accesses. So, we have

  $$22\ 000 + 2500 = 24\ 500$$

  block accesses in total.

- Now, we consider an index-based nested-loop join with department as the outer relation, we need

  $$b_D + r_D(x_{DNO} + s_{DNO} + 1) + (js \cdot r_E \cdot r_D)/f_{RS} = 12\ 288$$

  - Load a `DEPARTMENT` block.
  - Use the B+ Tree on `DNO` to find the s relevant blocks (and then the tuples) to join, where s is the selection cardinality of `DNO`.
  - Output the result.
  - Continue for all the `DEPARTMENT` blocks.

  Under the uniformity assumption, we find that the selection cardinality of `DNO` is

  $$s_{\texttt{DNO}} = 0.008 \cdot 10\ 000 = 80.$$

  So, the total number of blocks that we read is

  $$b_D + r_D(x_{\texttt{DNO}} + s_{\texttt{DNO}} + 1) = 9\ 788.$$

If we write the result back into the disc, this requires 2500 further block accesses. So, we have

$$22\ 000 + 2500 = 12\ 288$$

block accesses in total.

- Using a hash-join, we need

$$3(b_D + b_E) + (js \cdot r_E \cdot r_D)/f_{RS} = 8539$$

in total block accesses to read and write the blocks.

- Since we have a B+ Tree built on `DNO`, we cannot use the sort-merge algorithm. The attribute must be non-ordering.

So, the best approach to join them is a nested-loop join.

Example 4.3.2. We want to execute the following SQL query.

```
SELECT  *
FROM    EMPLOYEE E, DEPARTMENT D, DEPENDENT T
WHERE   T.E_SSN = E.SSN AND E.SSN = D.MGR_SSN;
```

We have

- $r_T = 50$ dependents in $b_T = 3$ blocks;
- $r_D = 125$ departments in $b_D = 13$ blocks;
- $r_E = 10\ 000$ employees in $b_E = 2000$ blocks;
- the blocking factor $f = 10$, and for the result block $f_{RS} = 2$;
- maximum $n_B = 100$ blocks in memory.

Under the uniformity assumption, there are

$$50/10 = 5$$

dependents per employee. Moreover, we have the following primary access paths:

- A clustering index on `T.E_SSN` of $x_{\text{E\_SSN}} = 2$ levels with 10 distinct `SSN` values.
- A primary index on `D.MGR_SSN` of $x_{\text{MGR\_SSN}} = 1$ level.
- A B+ Tree on `E.SSN` of $x_{\text{SSN}} = 4$ levels.

Find the number of block accesses require to compute the result using the following approaches:

1. First joining `EMPLOYEE` and `DEPENDENT`, and then joining the result with `DEPARTMENT`.
2. First joining `EMPLOYEE` and `DEPARTMENT`, and then joining the result with `DEPENDENT`.

1. In the first case, we first join `EMPLOYEE` and `DEPENDENT`. The join selectivity in this case is

$$js = 1/\max(10, 10\ 000) = 0.01\%.$$

So, we expect

$$jc = js \cdot |E| \cdot |D| = 50$$

tuples to be selected in this process.

For each dependent, we get their employee using the B+ Tree on `E.SSN`. This takes

$$b_T + r_T \cdot (x_{SSN} + 1) + jc/f_{RS} = 278$$

block accesses to read and write the tuples into memory. We can also use the clustering index on `T.E_SSN`. For each dependent, we get the dependent using this index. This takes

$$b_E + r_E \cdot (x_{\text{E\_SSN}} + \text{ceil}(s_{\texttt{E\_SSN}}/f)) + jc/f_{RS} = 32\ 025$$

block accesses to read and write the tuples into memory. Clearly, the better option is to use the B+ Tree.

In this case, the intermediate result has $r_{ET} = 50$ tuples, which fit in 25 blocks. These blocks can fit in memory. Next, we take the intermediate result and join it with the `DEPARTMENT` relation. So, we take a tuple from the intermediate result in memory and check whether it is a manager. The join selectivity for `EMPLOYEE` and `MANAGER` is

$$js = 1/\max(10\ 000, 125) = 0.01\%.$$

So, we expect

$$jc = js \cdot r_{ET} \cdot r_D = 0.625$$

tuples to be selected in the process. This will fit in 1 block.

We can use the primary index on `D.MGR_SSN` to filter out the tuples. We take a tuple from the intermediate result and use the primary index to retrieve the manager details. This requires

$$r_{ET} \cdot (x_{\text{MGR\_SSN}} + 1) = 100$$

blocks to be read from memory. We do not need to load the intermediate blocks since they are already in memory. Moreover, we do not write the final outcome into disks. In total, this plan requires

$$278 + 100 = 378$$

block accesses.

2. In the other plan, we first join `EMPLOYEE` and `DEPARTMENT`. Here, the join selectivity is

$$js = 1/\max(10\ 000, 125) = 0.01\%.$$

So, the join cardinality is

$$jc = js \cdot r_E \cdot r_D = 125$$

managers. Since the `DEPARTMENT` relation is sorted with respect to `MGR_SSN`, we can find all the managers in

$$\text{ceil}(jc/f_{RS}) = 63$$

blocks.

For each department, we can get its manager using the B+ Tree on `E.SSN`. This takes

$$b_D + r_D(x_{SSN} + 1) + jc/f_{RS} = 701$$

block accesses. Also, for each employee, we can get the department they manage (if any) using the primary index on `D.MGR_SSN`. This takes

$$b_E + r_E(x_{\text{MGR\_SSN}} + 1) + jc/f_{RS} = 22\ 063$$

block accesses. The better option here is to use the B+ Tree.

The intermediate result takes 63 blocks, which can fit in memory. We will now join this result with the `DEPENDENT` relation. The join selectivity for `EMPLOYEE` and `DEPENDENT` is

$$js = 1/\max(10\ 000, 10) = 0.01\%.$$

So, we expect

$$jc = js \cdot r_{ED} \cdot r_T = 0.625$$

tuples to be selected in the process. This will fit in 1 block.

We can take one intermediate tuple from memory, and check whether the manager has a dependent, using the clustering index on `T.E_SSN`. This requires

$$r_{ED}(x_{\text{E\_SSN}} + \text{ceil}(s_{\texttt{E\_SSN}}/f)) = 375$$

block accesses. In total, this plan requires

$$701 + 375 = 1076$$

block accesses.

Example 4.3.3. We want to execute the following query:

```
SELECT  *
FROM    EMPLOYEE E, DEPARTMENT D
WHERE   E.Salary = 1000 AND E.SSN = D.MGR_SSN;
```

There are $r_E = 10\ 000$ employee tuples in $b_E = 2000$ blocks. There are $r_D = 125$ department tuples in $b_D = 13$ blocks.
We have built the following primary access paths on the two relations:

- A clustering index over `E.Salary` of $x_{\text{Salary}} = 3$ levels. There are 500 distinct salary values.
- A B+ tree over `E.SSN` of $x_{\text{SSN}} = 4$ levels.
- A primary index over `D.MGR_SSN` of $x_{\text{MGR\_SSN}} = 1$ level.

The blocking factor of `E` and `D` is $f = 10$, and the joined tuple is $f_{RS} = 2$. We can fit $n_B = 100$ blocks in memory.
Find the number of block accesses required to compute the result using the following approaches:

1. First filtering and then joining.
2. First joining and then filtering.

1. We will first consider the filter-then-join approach. We can use the clustering index on `E.Salary` to find all the employees that satisfy `Salary = 1000`. Under the uniformity assumption, the selection selectivity is
$$sl(\text{Salary}) = 1/500 = 0.002.$$
So, the selection cardinality is
$$s_{\texttt{Salary}} = 0.002 \cdot 10\ 000 = 20.$$
That is, there are $r_{FE} = 20$ employees that satisfy this condition. Using the clustering index, we require
$$x_{\text{Salary}} + \text{ceil}(s_{\texttt{Salary}}/f) = 5$$
block accesses.
Next, we join the intermediate result with the relation `DEPARTMENT`. Since the intermediate result occupies 2 blocks, it can fit in memory. The join selectivity is
$$js = 1/\max(10\ 000, 125) = 0.01\%.$$
So, the join cardinality is
$$jc = js \cdot r_{FE} \cdot r_D = 0.25.$$
This fits in 1 block. For each tuple in the intermediate result, we use the primary index over `D.MGR_SSN` to get the relevant department tuple, if any. This requires
$$r_{FE} \cdot (x_{\text{MGR\_SSN}} + 1) = 40$$
block accesses. In total, we require
$$5 + 40 = 45$$
block accesses.

2. Next, we consider the join-then-filter approach. We will use the B+ Tree over `E.SSN` to join `EMPLOYEE` and `DEPARTMENT`. The selection selectivity is

$$js = 1/\max(10\ 000, 125) = 0.01\%.$$

Using the B+ Tree, we require

$$b_D + r_D \cdot (x_{\text{SSN}} + 1) + \text{ceil}(js/f_{RS}) = 701$$

block accesses. The intermediate result is 125 tuples (one for each department). This takes

$$\text{ceil}(125/f_{RS}) = 2$$

blocks. This can fit in memory.

Next, we filter the tuples from the intermediate result that satisfy `Salary = 1000`. This can be done by linearly scanning the entries in memory. It needs no further block access since the salary is already present in the intermediate result. So, it takes 701 block accesses in total.