Data Bases 2

Niccoló Didoni

September 2021

Contents

Part I

Transactional Systems

Chapter 1

Data Base Management Systems

A Data Base Management System (DBMS for short) is a software able to manage data that is

- **Large**, in fact the data managed by a DBMS can be much larger than the central memory.
- **Persisted**, in fact the data stored has remain the same in every execution (if not changed by the user).
- **Shared**, in fact the same data can be accessed by multiple users at the same time (e.g. a flight boarding app that get access to the same data base).
- **Reliable**, in fact the data does not have to be lost after every restart of the system.
- **Data Ownership Respectful** to assure privacy.

1.1 Architecture of a DBMS

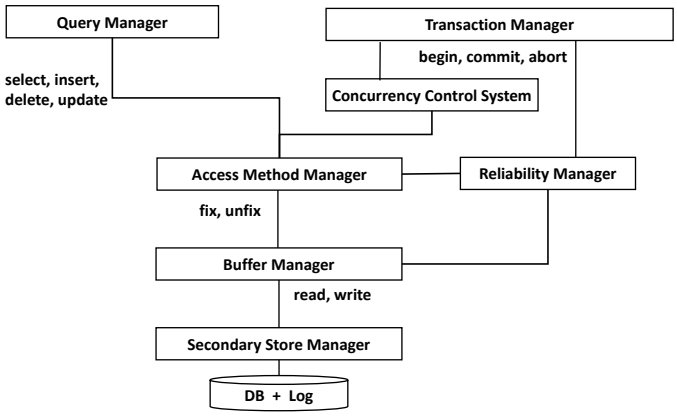

Figure 1.1: The architecture of a DBMS

1.2 Transactions

To ensure that all the properties of a DBMS are respected, relational data bases use **transactions**

Definition 1 (Transaction)**.** *A transaction is the basic unit of work of the DBMS that allows to execute a collection of instructions in an all or nothing fashion.*

This means that all the instructions contained in a transaction are either executed completely or not executed at all.

A transaction is encapsulated between the commands `BEGIN TRANSACTION` and `END TRANSACTION`.

```
BEGIN TRANSACTION
    -- transaction code
END TRANSACTION
```

1.2.1 Commit and rollback

A transaction can be committed or rolled back

- The transaction is committed via the `COMMIT-WORK` command. This command executes every operation of the transaction in the order in which they are written and the result is saved to memory.
- The transaction is rolled back via the `ROLLBACK-WORK`. If the transaction is rolled back all the commands executed are undone (the system goes back to the state before the transaction).

Both operations can be written in the same transaction but only one has to be executed (one could use an if sentence to decide whether the transaction has to be committed or rolled back).

1.2.2 Application

An application is a collection of transaction interlaced with other operations.

1.2.3 ACID properties

A transaction is

- **Atomic**.
- **Consistent**.
- **Isolated**.
- **Durable**.

Atomicity

Atomicity ensures that either every operation of the transaction is executed or no operation is executed, using the `COMMIT-WORK` and `ROLLBACK-WORK` commands respectively.

Atomicity is handled by the **Reliability Manager**.

Consistency

Consistency preserves the integrity of the data base. At the beginning and at the end of the transaction the state of the database has to consistent (the state can be checked using integrity constraints). The same doesn't applies to middle states of the transaction.

For instance let us consider a data base that has three tuples $A = 10, B = 50, C = 40$ and the sum of the three values has to be 100.

If we modify the database, at the end of the transaction, the sum of the three tuple has to be 100.

The consistency is handled by the **Integrity Control System** and the **Query Manager**.

Isolation

Isolation assures the correct execution of concurrent transactions, in fact the same transaction can be executed by multiple apps (e.g. a flight booking app in which many people might want to buy a ticket for the same flight) in a concurrent way. Isolation assures that the execution of concurrent transitions is the same as if they were executed serially in any order.

The isolation is handled by the **Concurrent Control System**.

Durability

The durability property assures that the result of the transaction will last forever after the transaction is over.

The durability is handled by the **Reliability Manager**.

Chapter 2

Concurrency Control Systems

The Concurrency Control System (CCS) is the component of a DBMS that handles concurrency and therefore assures the isolation property of a transaction.

Parallelism can be very efficient and can increase the number of transactions per second (TPS) executed but it also provides some challenges. A transaction can be divided in two parts

- One between the start of the transaction and the commit/roll-back phase.
- One between the commit/roll-back and the end of the transaction.

Let us focus on the first section. If the DBMS wants to execute two transactions T_1 and T_2 in parallel it can

- Run the transaction in sequence (not obtaining parallelism).
- Run the transaction **interleaved**, i.e. it interleaves commands of T_1 and T_2.
- Run the transaction **nested**, i.e. it runs some commands of T_1 than it stops, runs all the commands of T_2, and complete T_1.

2.1 Invalid sequences

Concurrent execution of multiple transactions can generate wrong results. Usually the errors are caused by misplaced reads and writes to memory. To analyse some invalid sequences of writes and reads we will use the notation

$$w_n(x) \quad r_m(y)$$

to indicate that the transaction n has written the variable x and that the transaction m has read the variable y, respectively.

When analysing the following errors (but one, the dirty read error) we will assume that a transaction is executed only if it has to be committed.

2.1.1 Lost update

A sequence

$$r_1(x), r_2(x), w_1(x), w_2(x)$$

brings to a lost update error because the transaction T_2 reads the same value of the transaction T_1 whilst it should read that value after T_1 has written it. In other words T_2 should read the value that T_1 wrote, not the same value T_1 read.

To highlight why this conflict can cause problems, let us consider two transactions (for brevity we will use the `Tn:` notation to represent the name of a transaction)

```
1  T1: UPDATE account
2      SET balance = balance + 3
3      WHERE client = '1053112';
4
5  T2: UPDATE account
6      SET balance = balance + 6
7      WHERE client_number = '1053112';
```

If the client has 100 as balance and the two transactions are executed serially, at the end the balance will be 109 (independently from the order in which the two transaction are executed). On the other hand if we want to use concurrency to speed up the execution it might happen that

1. First the balance is read (in both the transactions), so the value for balance is 100 in both cases.
2. Than the first transaction is executed and the value 103 is saved in memory.
3. Finally the second transaction is executed but, because the balance had been read before T_1 has saved its result, then T_2 will override the value 103 with 106. And the final result of the transactions will be 106 (not 109 as expected)

2.1.2 Dirty read

A dirty read error can occur when a transaction T_2 reads a value written by another transaction T_1 before the transaction T_1 has committed. During a transaction the DBMS cannot know if, in the end, the transaction has to be committed so it has to save to memory anyways. If the transaction is rolled back than the memory is brought back to its original state. An example of erroneous sequence is

$$r_1(x), w_1(x), r_2(x), abort_1, w_2(x)$$

This sequence generates an erroneous result because the operation $r_2(x)$ reads the value written by $r_1(x)$, but this operation has to be rolled back, so the value written by $w_1(x)$ should not be considered. Notice that this is a conflict because when T_1 aborts, T_2 can continue its execution.

2.1.3 Non repeatable read

A non repeatable read error occurs when a transaction T_1 needs to read the same value twice. In this case if a transaction T_2 reads the value and immediately writes it before T_1 reads it the second time, the second read of T_1 will retrieve a wrong value. An error of this kind is represented by a sequence

$$r_1(x), r_2(x), w_2(x), r_1(x)$$

Say that at the beginning $x = 100$. Then both T_1 and T_2 read $x = 100$. After that T_2 writes $x \leftarrow x + 100$. At the end T_1 reads a different value ($x = 200$) than the one read at the first step ($x = 100$), thus a conflict has happened.

2.1.4 Phantom update

A phantom update error occurs when two transactions T_1 and T_2 work on a same set of values that has to satisfy a constraint. For instance let say we have a set of values A, B and C whose sum has to be 100. T_1 could read A and B and then T_2 could read and overwrite A, B and C. Even if T_2 has preserved the constraints, the values A and B read by T_1 do not satisfy the constraint so the result of the operation done by T_1 wont satisfy the constraints.

$$r_1(A), r_1(B), r_2(A), r_2(B), r_2(C), w_2(A), w_2(B), w_2(C), r_1(C), w_1(A), w_1(B), w_1(C)$$

To give a more practical example say that

$$A = 30, B = 10, C = 60$$

and let assume that T_1 subtracts 10 from C and adds 10 to B whilst T_2 subtracts 10 from A and adds 10 to B.

```
1     T1: UPDATE account
2         SET C = C - 10;
3         SET B = B + 10;
4
5     T2: UPDATE account
6         SET A = A - 10;
7         SET B = B + 10;
```

The execution would be as follows

1. The data base satisfies the constraint $(A + B + C = 100)$
2. T_1 reads $A = 30$ and $B = 10$
3. T_2 reads $A = 30$, $B = 10$ and $C = 60$
4. T_2 writes $A = 30$, $B = 20$ and $C = 50$
5. T_1 reads $C = 50$ but now the state of T_1 is inconsistent because $A + B + C = 30 + 10 + 50 = 90 \neq 100$
6. T_2 writes $A = 20$, $B = 20$ and $C = 50$ which is an inconsistent state.

2.1.5 Phantom insert

A phantom insert error occurs when a transaction T_1 has to execute an aggregate function twice and between the two operations another transaction T_2 inserts a tuple that respects the conditions to be included in the tuples on which the aggregated function is executed. In other words after the insertion the aggregate function will have two different results.

Consider a table

```
MARKS(student_name, sex, mark);
```

containing the following tuples

```
'Luke' 'M' 8
'Anna' 'F' 9
```

If transaction T_1 computes twice the average of every male student and a transaction T_2 adds a row with another male student between the two operations, than the second average (the one computed after the insertion) differs from the first one.

2.2 Scheduler

The scheduler is the component that allows concurrent execution without generating errors. The main job of the scheduler is to discard the sequences of operations that generate an anomaly (for example the scheduler discards the sequence r_1, r_2, w_1, w_2 because it generates a lost update error). To do its job a scheduler has to accept or reject the operations requested by the transactions.

The sequences of operations (also called **schedules**) analysed are composed of the operations of the transactions that have to executed in parallel. More precisely

Definition 2 (Schedule)**.** *A schedule is a sequence of operations performed by concurrent transactions* ***that respects the order of operations of each transaction****.*

In other words a schedule is a permutation of the operations of every transaction in which the operations must be in the same order of the transaction they refer to. For instance if $T_1 = r_1(x), r_1(y), w_1(x)$ and $T_2 = r_2(x), w_2(x)$ then

$$r_1(x), r_2(x), r_1(y), w_2(x), w_1(x)$$

is a schedule (even if it generates an error) because the the operations are in the same order of the original transaction whilst

$$r_1(x), w_2(x), r_1(y), r_2(x), w_1(x)$$

isn't a schedule because $w_2(x)$ comes before $r_2(x)$, contrary to the order of the transaction T_2 in which $r_2(x)$ is executed before $w_2(x)$.

Serial sequences

The sequences in which the transactions are executed in sequence are called serial sequences (or **serial schedules**).

If we consider n transactions then there exists $n!$ different serial schedules.

$$N_{serial} = n!$$

For instance the schedules T_1, T_2 and T_3 can be arranged as $T_1T_2T_3$, $T_2T_3T_1$, $T_3T_1T_2$, $T_3T_2T_1$, $T_1T_3T_2$ and $T_2T_1T_3$ (and $3! = 6$, as expected).

On the other hand if we consider that every transaction T_i has k_i operations then we have

$$N_d = \frac{(\sum_{i=1}^{n} k_i)!}{\prod_{i=1}^{n} k_i!}$$

different schedules. In particular N_d is the number of permutations of all operations divided by the product of the number of permutations of the operations of each transaction (only 1 out of the k_i permutations is valid, the one that respects the sequence of operations).

Class of equivalence

Serial sequences are important because they always generate a valid result. The scheduler can therefore check if a certain sequence produce the same result as one of the serial sequence. In this case the sequence is valid, otherwise it is rejected.

The sequences that produce the same result are equivalent and therefore part of the same class of equivalence.

Methods to verify equivalence

After defining what equivalence means, we have to find methods that can check if a sequence is equivalent to a serial sequence (i.e. if it belongs to the same class of equivalence) without checking all the $n!$ possibilities (which would be too slow).

2.2.1 View serialisability

The first method used to check if a certain schedule is valid (i.e. it is equivalent to a serial schedule) is based on two properties

- **Reads from**. A read operation $r_i(x)$ reads from a write operation $w_j(x)$ if $w_j(x)$ precedes $r_i(x)$ and there is no $w_k(x)$ between $r_i(x)$ and $w_j(x)$.
- **Final write**. A write operation $w_i(x)$ is a final write if it is the last write of the variable x in the schedule.

View equivalence

Thanks to this properties we can say that two schedules S_a and S_b are view-equivalent if

1. S_a and S_b have the same operations.
2. S_a and S_b have the same reads from relationships.
3. S_a and S_b have the same final writes.

and we write

$$S_j \approx_V S_i$$

View serialisability

Given the definition of view equivalence we can say if a schedule is view serialisable. In particular

Definition 3 (View serialisability)**.** *A schedule S is view serialisable if S is view equivalent to a serial schedule made of the same transactions (in any order).*

The class of view serialisable schedules is called VSR and strictly contains the class of serial schedules as shown in Figure 2.1.

Notice that if the transactions of a schedule read or write different variables, we should define the read-from and final-write relationships on all variables independently and then find a serial schedule that satisfies the relationships on all variables at the same time.

For example let $T_0 = w_0(x)$, $T_1 = r_1(x), w_1(y), w_1(x)$ and $T_2 = r_2(x)$. The schedule $S_1 = w_0(x), r_1(x), r_2(x), w_1(y), w_1(x)$ is view equivalent to serial schedule $S_s = r_2(x)r_1(x), w_1(y), w_1(x)$ because

- $w_1(x)$ is the last write in both cases.
- the reads from relationships $r_1(x)$ reads from $w_0(x)$, $r_2(x)$ reads from $w_0(x)$ are the same in both schedules.

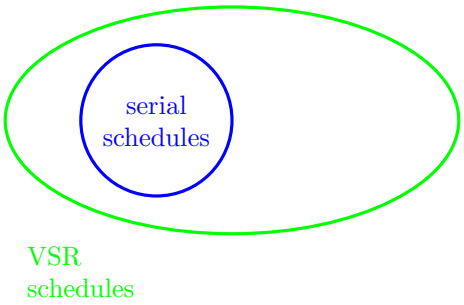

Figure 2.1: VSR class

Because S_1 is view equivalent to a serial schedule, then S_1 is view serialisable (i.e. $S_1 \in VSR$).

Consider now the transactions $T_0 = w_0(x)$, $T_1 = r_1(x), w_1(x)$ and $T_2 = r_2(x), w_2(x)$. The schedule $S_2 = w_0(x), r_1(x), r_2(x), w_1(x), w_2(x)$ is not view serialisable because it exists a serial schedule in which $w_2(x)$ is the final write but we cannot find a serial schedule in which both $r_2(x)$ and $r_1(x)$ read from $w_0(x)$ (not considering T_0, the only two serial schedules are $w_0(x), r_1(x), w_1(x), r_2(x), w_2(x)$ and $w_0(x), r_2(x), w_2(x), r_1(x), w_1(x)$).

Problem complexity

Checking if two schedules are view equivalent can be done in polynomial time (n^2 time complexity).

Things change when we have to find if a schedule is view serialisable. In this case we have to check the equivalence with every possible serial schedule. This problem is therefore NP-complete (it cannot be done in polynomial time), since there exists $n!$ serial schedules with n transactions.

2.2.2 Conflict serialisability

The conflict serialisability is based on two types of conflict

1. The conflict between **read and write operations** (i.e. $r(x) - w(x)$ and $w(x) - r(x)$).
2. The conflict between **two write operations** (i.e. $w(x) - w(x)$).

Conflict equivalence

Two schedules S_1 and S_2 are conflict equivalent if

1. They have the same operations.
2. In all the conflicting pairs the transactions occur in the same order.

and we write

$$S_1 \approx_C S_2$$

Conflict serialisability

Given the definition of conflict equivalence we can say if a schedule is conflict serialisable. In particular

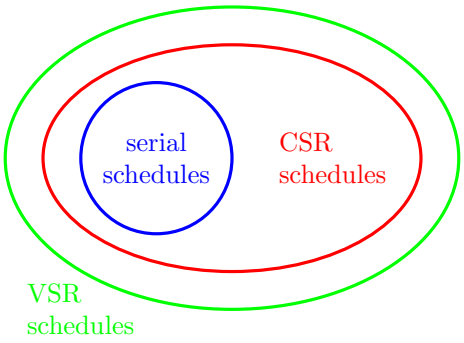

Figure 2.2: CSR class.

Definition 4 (Conflict serialisability)**.** *A schedule S is conflict serialisable if S is conflict equivalent to a serial schedule made of the same transactions (in any order).*

The equivalence class of conflict serialisable schedules is called CSR. The equivalence class CSR is a subset of the equivalence class of view serialisable schedules VSR

$$CSR \subset VSR$$

In other words all conflict serialisable schedules are also view serialisable, but the converse is not necessarily true. We can also say that conflict equivalence implies view equivalence.

$$\text{conflict equivalence} \Rightarrow \text{view equivalence}$$

Conflict graph

To test conflict serialisability of a schedule S we can use a conflict graph in which

- The nodes of the graph represent the transactions of S.
- If there exists a conflict between two operations (on the same resource x) $o_i(x)$ and $o_j(x)$ of transactions T_i and T_j than we must draw an arc from T_i to T_j if o_i precedes o_j, from T_j to T_i otherwise.

A schedule S is conflict serialisable if and only if the conflict graph is acyclic. For instance the schedule

$$S = w_0(x), r_1(x), w_0(z), r_1(z), r_2(x), w_0(y), r_3(z), w_3(z), w_2(y), w_1(x), w_3(y)$$

generates the conflict graph 2.3 which is not cyclic and therefore S is conflict serialisable.

Conflicts should be considered only on the same variable but the conflict graph must contain the conflicts on all resources (i.e. variables).

When checking if the graph is acyclic (we can iteratively remove a node that has no incoming arcs until no node remains) we also build a topological (partial) order (i.e. the order in which the nodes are removed). This order represents the order of the transitions of the serial schedule to which S is equivalent to. In our example the topological order is $T_0 \rightarrow T_2 \rightarrow T_1 \rightarrow T_3$, thus S is conflict equivalent to the serial schedule $T_0T_2T_1T_3$.

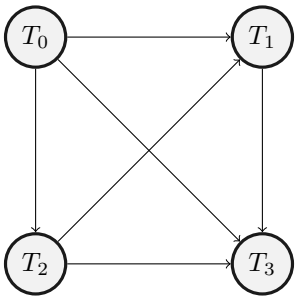

Figure 2.3: Conflict graph for the schedule S

Blind writes

A write operation $w_i(x)$ is blind if

- it is not the last operation on resource x and
- the following operation on x is a write $w_j(x)$.

Blind writes are important because each schedule S that is view serialisable (i.e. $S \in VSR$) but not conflict serialisable (i.e. $S \notin CSR$) has, in its conflict graph, cycles of arcs due to pairs of blind writes. These can be swapped without modifying the reads-from and final write relationships. Once the graph is acyclic it is possible to find, using topological sorting, a serial schedule view-equivalent to the initial one .

2.3 2 Phase locking

Both conflict and view serialisability require the model to know if a transaction commits before the execution. This is a problem because we would like to abort a transaction while it is executing. For this reason conflict and view serialisability are not feasible and are just theoretical models. One method used to decide if a certain sequence doesn't generate anomalies during execution is locking.

Locks In this method every operation has to ask a lock (like in multi-threading programming). Locks are divided in

- **Read locks**. Read locks, also known as shared locks, are requested by a read operation.
- **Write locks**. Write locks, also known as exclusive locks, are requested by a write operation.
- **Update locks**. When a transaction needs to read and than to write the same data, it can ask for an update lock. It can be used to read the data but after that the transaction has to get a write lock to write it. The difference with a read lock is that it is not shareable, i.e. when a transaction gets an update lock on a resource, no other transaction can get a lock on that same resource. An update lock can be asked using the `SELECT FOR UPDATE` statement in SQL.
- **Predicate locks**. Predicate locks allow to place a lock on a predicate, that is on every tuple that satisfy a predicate (i.e. a condition). This type of lock prevents other transactions to insert, delete or update any tuple satisfying the predicate.

All locks can be released with an **unlock** command.

2.3.1 State of objects

A lock can block some actions on a certain object. For instance the command $w_1(x)$ requests a write lock on the object x. For this reason an object can find itself in three different states

1. An object is **free** if no operation has asked for a lock.
2. An object is **write locked** if a write operation has asked for a lock. A transaction can take a write lock on an object only if
   - It is the only transaction to have a read lock on such object (i.e. all other transactions have unlocked the object).
   - The object is free.
3. An object is **read locked** if one or more read operations have asked for a lock. Multiple operations can ask for a read lock on the same object so the object also has a counter n that stores the number of transactions that are currently read-locked to that object.
4. An object is **predicate locked** if the lock is not on some specific tuples but it's on the tuples that satisfy a predicate. This lock is used to prevent phantom inserts.

When a new operation arrives to the Concurrency Control System, depending on the type of operation, the CCS can modify the state of an object. Table 2.1 shows all the possible combination of incoming command, current and next state.

Request	Free	Read locked	Write locked
read lock	OK read locked	OK read locked (n++)	WAIT write locked
write lock	OK write locked	WAIT read locked	WAIT write locked
unlock	error	OK depends(n - -)	OK free

Table 2.1: All combination of incoming commands and states.

When a transaction tries to ask for a lock that cannot be given, such transaction is putted in a wait state, in fact it has to try to get the lock again in a second time.

2.3.2 Upgrade and downgrades

A transaction can upgrade and downgrade its locks. In particular

- If a transaction has a read lock on a resource, and it can obtain a write lock then the transaction can upgrade its read lock to a write lock.
- If a transaction has a write block on a resource, then it can downgrade such lock to a read lock without releasing it.

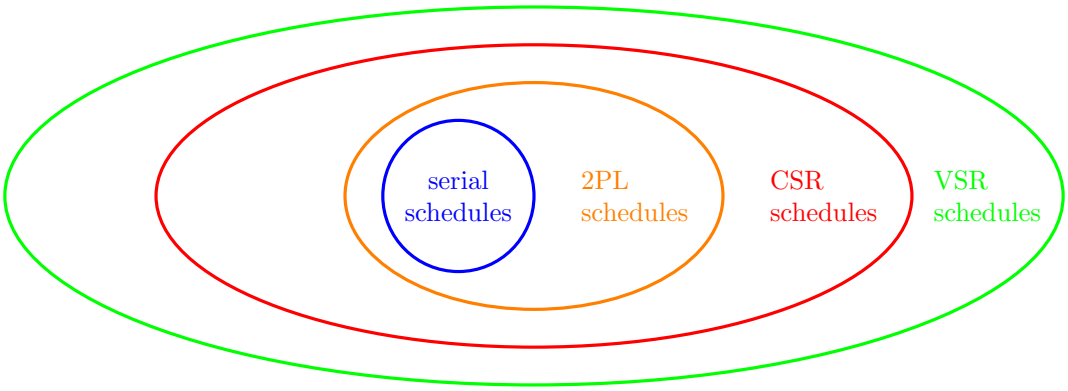

Figure 2.4: 2PL class.

Unlock problem

In theory an unlock could occur anytime after a certain operation is over (even before the end of the transaction).

This behaviour can lead to non repeatable reads, in fact there could be a sequence such that

1. A transaction T1 wants to read x ($r1(x)$). The object x is free so T1 acquires the read lock.
2. After the read operation is over T1 releases the lock with an unlock operation.
3. Now a transaction T2 can read and write x because the lock for x has been released by T1. After writing T2 releases the lock on x.
4. If T1 wants to read again, it can do it because T2 has released the lock, but T1 reads a different value with respect to the first read because T2 has (legally) overwritten it.

To solve this problem we have to introduce a new rule. A transaction **can't acquire a lock after releasing one** (even on different object). The result of this rule is that all transactions have to acquire all the locks (not necessarily together) before one lock is released. Notice that

- Downgrading a write lock to a read lock counts as releasing the lock.
- Upgrading a read lock to a write lock counts as obtaining a lock, thus all locks have to be upgraded (if necessary) before releasing the first lock.

Locking schedulers that satisfies this rule are called 2 Phase Locking schedulers. The class of equivalence of such schedulers is called 2PL.

The class 2PL is strictly contained in the class of conflict serialisable schedulers (hence a 2PL schedule is also conflict and view serialisable)

$$2PL \subset CSR \subset VSR$$

For this reason the class of 2PL schedulers accepts less schedules (sequences) that the other schedulers but it can be used in real life applications because it can be applied to real-time sequences.

Growing and shrinking phase

To better understand how locks have to be acquired and released we can define

- A **growing phase**. During the growing phase a transaction can acquire and upgrade locks. In this phase locks can't be released or downgraded, in fact downgrading a locks means releasing a write lock. The growing phase of a transaction starts when the first lock is acquired and ends when the last lock is acquired.
- A **shrinking phase**. During the shrinking phase a transaction can release and downgrade locks. In this phase locks can't be acquired or upgraded. The releasing phase starts when the first lock is released and ends when all locks have been released.

An representation of the growing and shrinking phases is shown in Figure 2.5. This representation refers to a transaction and considers all the variables on which the transaction works. Notice that the shrinking phase can start before the transaction commits.

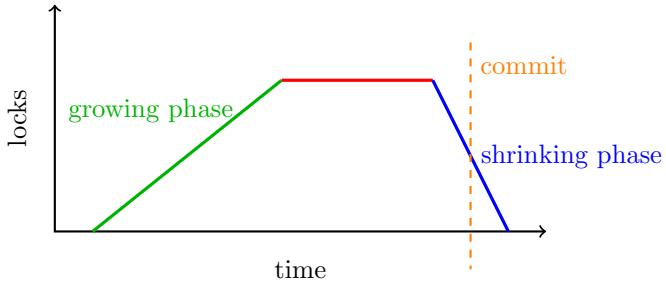

Figure 2.5: Growing and shrinking phases for 2PL.

2.3.3 Strict 2PL

Two Phase Locking is still vulnerable to dirty reads, in fact locks can be released before a transaction commits. To solve this problem we can further restrict 2PL and impose to release locks only after the commit operation. This new scheduler is called Strict 2PL scheduler and the class of schedules accepted by a Strict 2PL scheduler is called Strict 2PL.

The locks in Strict 2PL are called **long duration locks** (whilst 2PL locks are called short duration locks) because are released all together at the end of the transaction. This behaviour can lead to less performance and longer waiting times but it solves all the anomalies. Growing and shrinking phases of 2PL-strict are represented in Figure 2.6. We can notice that the locks are released all at once since the shrinking phase is a vertical line.

Level of locking in real life systems

In real life applications we can specify some isolation levels depending on the types of anomalies we allow to happen. In particular SQL allows the following isolation levels

- **READ UNCOMMITTED** allows an object to be read even if not committed. This level solves only the lost update anomaly. With this isolation level no read lock is used (and locks of other transactions are ignored).

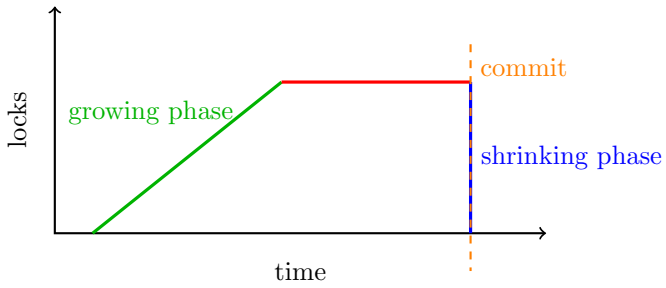

Figure 2.6: Growing and shrinking phases for 2PL-strict.

- **READ COMMITTED** allows only to read committed data (read locks are short duration locks and can be obtained even after being released, unlike 2PL). This level solves the dirty read anomaly but allows non-repeatable reads, phantom inserts and phantom updates.
- **REPEATABLE READ** allows only long duration read locks. This level solves dirty reads, non-repeatable reads and phantom updates and allows only the phantom insert anomaly.
- **SERIALIZABLE** that is equivalent to Strict 2PL with predicate locks. This level solves all anomalies. Notice that a serialisable schedule isn't a serial schedule (i.e. the transaction aren't executed one after the other without mixing their operations). A schedule is serialisable if it gives the same result as a serial schedule.

The write lock level cannot be changed. A summary of the locking levels is shown in Table 2.2 and 2.3.

To specify the isolation level we can use the following syntax

```
1    <set transaction statement> ::= SET [ LOCAL ]
2    TRANSACTION <transaction characteristics>
3
4    <transaction characteristics> ::= [ <transaction mode> [ { <comma>
5  <transaction mode> }... ] ]
6
7    <transaction mode> ::= <isolation level> | <transaction access mode>
8                                             | <diagnostics size>
9
10   <transaction access mode> ::= READ ONLY| READ WRITE
11
12   <isolation level> ::= ISOLATION LEVEL <level of isolation>
13
14   <level of isolation> ::= READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ |
15                           SERIALIZABLE
```

	Dirty read	Non repeatable read	Phantom reads
Read uncommitted	Yes	Yes	Yes
Read committed	No	Yes	Yes
Repeatable read	No	No	Yes (insert)
Serialisable	No	No	No

Table 2.2: Anomalies allowed for each isolation level

	Read locks	**Write locks**
Read uncommitted	Not required	Well formed writes Long duration write locks
Read committed	Well formed reads Short duration read locks (data and predicate)	Well formed writes Long duration write locks
Repeatable read	Well formed reads Long duration data read locks Short duration predicate read locks	Well formed writes Long duration write locks
Serialisable	Well formed reads Long duration read locks (predicate and data)	Well formed writes Long duration write locks

Table 2.3: Locks needed for each isolation level

2.3.4 Update locks

One of the most frequent situation in which a deadlock occurs is when two transactions T_1 and T_2 acquire a read lock for a resource and then one of them tries to update the lock to a write lock (that has to be exclusive to the transaction that has to write).

To solve this problem a transaction can use an **update lock** that allows the transaction to read and write a resource. If a resource R is update locked, the other transactions can only read lock R, until the update lock isn't removed by the owning transaction.

Update locks can be asked in SQL with the keyword `SELECT FOR UPDATE`.

2.3.5 Hierarchical locking

Until now we said that a transaction can lock a generic resource without specifying the nature of such resource. In principle a transaction could lock an entire table or a single tuple. In the former case concurrency is limited because even if the locking transaction is using a few rows, it is blocking other transactions from using other tuples without actual conflicts. In the latter case a transaction might need to acquire too many locks and reduce performance.

Long story short, we should find a way to lock the exact amount of resource needed with the best granularity possible (e.g. schema, table, tuple, entity).

Hierarchical locking can be used to achieve this goal. Hierarchical locking locks resources top-down starting from the lowest level of granularity (i.e. the schema) and going down until the right granularity level is obtained. Vice-versa, this type of locking releases locks bottom-up, starting from the locked level and going to the lowest levels of granularity.

Notice that high granularity means low hierarchical level because a resource with high granularity is very specific, thus on the lower levels of the hierarchy.

Locks

Hierarchical locking uses five types of locks

- **Read** locks (SL).
- **Write** locks (XL).
- **Intention Shared Lock**. An ISL expresses the intention of read-locking a resource with an higher level of granularity (i.e. lower in the hierarchy). An Intention Shared Lock can be

Request	free	ISL-locked	IXL-locked	SL-locked	SIXL-locked	XL-locked
ISL	Yes	Yes	Yes	Yes	Yes	NO
IXL	Yes	Yes	Yes	NO	NO	NO
SL	Yes	Yes	NO	Yes	NO	NO
SIXL	Yes	Yes	NO	NO	NO	NO
XL	Yes	NO	NO	NO	NO	NO

Table 2.4: Hierarchical locking table.

shared, this means that if a resource is Intention Shared Locked by T_1, another transaction T_2 can ISLock the same resource.

- **Intention Exclusive Lock**. An IXL expresses the intention of write-locking a resource with an higher level of granularity. An Intention Exclusive Lock can be shared, this means that if a resource is Intention Exclusive Locked by T_1, another transaction T_2 can IXLock the same resource.
- **Shared Intention Exclusive Lock**. A SIXL locks a resource in read mode with intention of locking a sub-resource in write mode. For instance a transaction can lock a table in read mode and express the intention of locking a row in write mode.

A transaction T can

- Get a read lock or a ISL on a non-root element only if T holds an equivalent or more restrictive (i.e. a ISL or IXL) on the parent node. For instance T can request a SL lock on a tuple only if it has a SL on the table in which the tuple is.
- Get an XL, IXL or SIXL lock on a non-root element only if T has an equally or more restrictive lock (SIXL or IXL) on its parent.
- Lock a resource based on the rules specified in the hierarchical lock granting table (Table 2.4).

2.4 Deadlocks

When a transaction can't acquire a lock it is suspended (i.e. it goes to a wait status). Waiting transactions are handled with a queue (i.e. FIFO). For this reason when we use locks to handle concurrent access we always have to take care of deadlocks (i.e. transactions that mutually wait each other) and starvation. For instance two transactions

$$T1 : r1(x), w1(y)$$

and

$$T2 : r2(y), w2(x)$$

could lead to a deadlock if we consider a sequence $r1(x), r2(y), w1(y), w2(x)$ because

1. T1 and T2 acquire the read lock on x and y respectively.
2. T1 can't acquire the write lock on y because T2 hasn't unlocked the resource (T2 has to take the write lock on x before releasing y), thus T1 is suspended. T2 should anticipate locking x before $w_1(y)$ (to release y and allow T1 to lock y) but it can't because T1 can't release the read lock on x (T1 is waiting T2 on releasing y before releasing x).

3. T2 can't acquire the write lock on x because T1 (that has only been suspended, not aborted, thus still has the read lock on x) hasn't unlocked the resource. T2 is suspended.

As we can see T1 and T2 wait each other and for this reason generate a deadlock. Handling deadlocks is fundamental to ensure good performance when executing concurrent transactions. In particular we can

- Set **timeouts** to solve deadlocks, in fact if a sequence generates a deadlock the execution will stop for a long time. If such time exceeds the timeout both or just one of the transaction are aborted to allow the other transactions to continue their execution. The timeout length doesn't have to be too short nor too long (poor performance).
- Use **heuristics to prevent** deadlocks. Such heuristics do not know if a situation causes a deadlock but if they found a sequence that may cause one they immediately abort both or just one of the transactions.
- Use **wait-for graphs to detect** deadlocks and kill the transactions that are in a deadlock. Notice that in this case we are detecting (not preventing) a deadlock, in fact we kill a transaction is killed when we are sure that a deadlock is occurring.

2.4.1 Deadlock detection

Lock graphs

A lock graph is a bipartite graph used to represent the transitions that request or hold some resources. In particular

- A node can be either a resource or a transaction (graphically represented in two different ways).
- An arc from a transaction T to a resource R represents the fact that T holds or requests (represented on the arc label) resource R.

Wait-for graphs

To discover deadlocks before they occur we can use a wait-for graph in which

- Every node corresponds to a transaction.
- An arch from T_a to T_b represents that the transaction T_a waits for an object locked by T_b.

If the graph is cyclic the sequence leads to a deadlock. When a scheduler finds a deadlock it rolls back the transaction (and eventually the application can decide to execute the transaction again to try another sequence that does not generate a deadlock).

Timeouts

Wait-for graphs aren't the only method that can be applied to detect deadlocks, in fact we can also use **timeouts**. In particular if a sequence generates a deadlock the execution will stop for a long time. If such time exceeds a timeout all or just some of the transactions are aborted to allow the other transactions to continue their execution. A critical part in using timeouts is deciding how long the timeout has to be, in fact

- If the timeout is too long the system's performance get worst.
- If the timeout is too short we might kill and restart many transactions, thus worsening performance.

2.4.2 Deadlock prevention

Deadlocks can also be prevented, in particular we can use some heuristics to understand if a schedule could bring to a deadlock and abort the transactions that might generate the deadlock. Notice that in this case we are not sure that a deadlock has occurred (like when detecting a deadlock with wait-for graphs and timeouts). Deadlock can be prevented with

- **Resource-based prevention**. Resource-based prevention requires the transactions to request all locks at once and only once. The main problem of this heuristic is that it's be hard for transactions to anticipate all locking requests.
- **Transaction-based prevention**. Transaction-based prevention assigns an identifier to each transaction. The identifier represents the age of the transaction. Ids are used to prevent older transactions from waiting younger ones. When an older transaction waits for a younger one either
  - The holding transaction is killed (**preemtive**, wound-wait).
  - The requesting transaction is killed (**non-preemptive**, wait-die).

2.4.3 Obermarck's algorithm

Obermarck's algorithm can detect a deadlock (i.e. a cycle in a wait-for-graph). This algorithm is based on the following assumptions:

- The transactions execute on a single main node.
- The transactions may be decomposed in sub-transactions that can be executed on other nodes.
- When a transaction spawns a sub-transaction it (the mother transaction) suspends its work until the sub-transaction completes.
- There can only be two types of wait-for relationships
  - T_i waits for T_j on the same node because T_i needs some data locked by T_j.
  - A sub-transaction of T_i waits for another sub-transaction of T_i running on a different node.

The Obermarck's algorithm tries to detect deadlocks without building every time the graph. In other words every node doesn't have to build a representation of all the other nodes.

Distributed dependency graph

Before understanding how the Obermarck's algorithm detects deadlocks without a global view, we have to build the global view to get the bigger picture. Let us assume that a transaction $T3$ starts on node B and it spawns some sub-transactions in the same node B and other sub-transactions in another node A. A representation of such global view is shown in Figure 2.7 (ignore the red dashed line between T_{2b} and T_{1b}).

In the graph

- An arc between two transactions on a node (square transactions) represents a **wait-for relation** like in wait-for graphs (i.e. $T \to T'$ means that T waits for T').

$$T \to T' \longmapsto T \text{ waits for } T'$$

- An arc between a transaction T and a call to another node N (a red round node) represents the fact **that the source T is waits for a transaction on another node** (e.g. in the graph 2.7 $T_{1a} \to E_B$ means that T_{1a} calls and waits for a transaction on remote node B, in particular T_1b).

$$T_{1a} \to E_B \longmapsto T_{1a} \text{ calls and waits for remote transaction } T_{1b}$$

- An arc between a call node N (a red round node) and a transaction T **represents the fact that the calling transaction is waiting for a transaction on another node** (e.g. in the graph 2.7 $E_B \to T_{2a}$ means that a transaction on remote node B, i.e. T_{2b}, is waiting for transaction T_{2a} in A).

$$E_B \to T_{2a} \longmapsto \text{ transaction on remote node } B \text{ is waiting for local transaction } T_{2a}$$

Communication protocol

Analysing the complete graph is simple but we want to find a way to detect locks only knowing the internal view of one of the nodes. To achieve such goal we have to communicate with the other nodes to obtain a local projection of the global dependencies. Let us focus on one node (say node A, but the same reasoning is true for B). Node A sends its local info to a node B only if both the following conditions apply

- A contains a transaction T_i that is waited for from another remote transaction and waits for a transaction T_j active on B.
- $i > j$, this ensure some sort of message forwarding along a node path where node A precedes node B if $i > j$.

In our example T_{2a} is waited for by T_{2b} on remote node B and A waits for T_{1b} on node B. Since $i = 2$ (because the waited transaction on A is T_{2a}) is greater than $j = 1$ (because the waited transaction by A is T_{1b}), thus A sends its local info to B.

Practically node A sends info to B if a distributed transaction listed at A waits for a distributed transaction listed at B with smaller index.

$$A \text{ sends info to } B \iff E_B \to T_{iA} \to \cdots \to T_{jA} \to E_B \wedge i > j$$

Algorithm The algorithm consists of four steps that periodically run on each node

1. Get graph info (wait dependencies among transactions and external calls) from the previous nodes. Sequences contain only node and top-level transaction identifiers.
2. Update the local graph by merging the received information.
3. Check the existence of cycles among transactions denoting potential deadlocks: if found, select one transaction in the cycle and kill it.

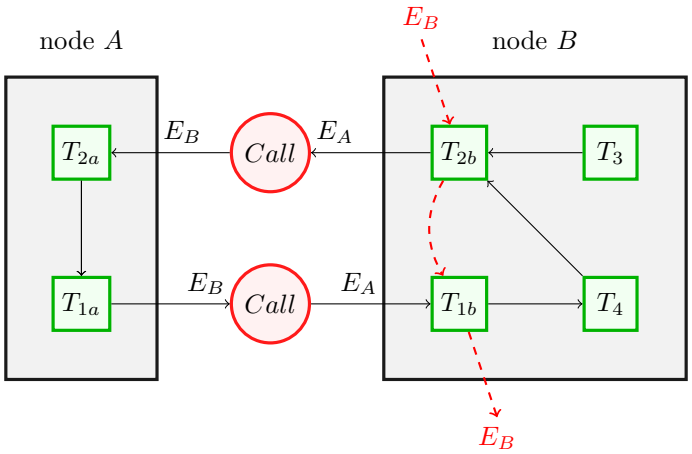

Figure 2.7: Global view of a distributed transaction.

4. Send updated graph info to the next nodes. Notice that in this phase we have to consider the info received at step 1.

In our example

1. A sends the info $E_B \to T_{2a} \to T_{1a} \to E_B$ to B.
2. B updates its graph adding a wait-for dependency between T_{2b} and T_{1b} (the red dashed arc) and the calls $E_B \to T_{2b}$ and $T_{1b} \to E_B$.
3. B detects a cycle between T_{2b}, T_{1b} and T_4, thus a deadlock may occur. To prevent the deadlock B has to kill either T_1 or T_2 or T_4.

Variations

The Obermarck's algorithm can detect deadlocks in four different ways, in particular a node N

- Send a message to the following node (i.e. to the node that N is waiting for) when $i > j$.
- Send a message to the preceding node (i.e. to the node that waits for N) when $i > j$.
- Send a message to the following node (i.e. to the node that N is waiting for) when $i < j$.
- Send a message to the preceding node (i.e. to the node that waits for N) when $i < j$.

Probability of deadlocks

The probability to have a deadlock depends on the number of transactions, in fact the higher is the number of transactions the higher it is the probability of a deadlock. The probability also depends on the number of operations for each transaction. In particular the probability of having a deadlock is less than the probability of having a deadlock (which is quadratic). It is more probable to have

2.5 Timestamps

Concurrency control can also be achieved using timestamps. Timestamps are an optimistic method, in fact in this case we assume that no conflict will occur. In other words a conflict is more of an exceptional situation compared to the usual execution on a database. For this reason conflicts are handled after they occur (and not in advance like for 2 phase locking).

Timestamps Optimistic concurrency control methods are based on timestamps, in fact each transaction has a timestamp that is assigned at the beginning of the execution. Using a timestamp for each transaction allows us to sort the transactions by age. For instance the first transaction that starts its execution is called T_0 (where 0 is the timestamp), the second is called T_1 and so on.

Schedule A schedule can be accepted if the concurrent execution of transactions is equivalent to the serial execution of the transactions according to their timestamps. In other words the only serial schedule accepted is the one in which the transactions are ordered by their age (i.e. using the timestamps).

Timestamps assignment On a single machine it's easy to assign a timestamp to each transaction. The same isn't true for distributed systems in which there isn't a global time.

To assign a timestamp in a distributed system we can use the Lamport's algorithm that states that a timestamp can be formed using a pair of identifiers

- An event id, which is the local timestamp of the machine.
- A node id which is an unique id, different for each node.

```
timestamp = event_id.local_id
```

The idea is that each machine increases its timestamp and when it receives a timestamp from another machine, it synchronises the local timestamp counter. For instance if the local counter of the machine 2 is 3 (i.e. the next timestamp will be 3.2) and a timestamp 5.1 arrives from the machine 1, then the machine 2 has to update its counter to 5 (basically the machine has received a message from the future because her counter was at 3 but the message received had counter of 5, when the machine updates the value to 5, it syncs itself with the machine 1). After the synchronisation, the next timestamp of the machine 2 will be 5.2.

2.5.1 Scheduler counters

The scheduler keeps two counters for each resource x

- The read timestamp (RTM).
- The write timestamp (WTM) that represents the timestamp of the last transaction that has written on the resource x.

When a new operation has to be executed the counters are updated and are used to decide if the operation can be executed. In particular when a read operation $r_{ts}(x)$ arrives

- If the timestamp of the request is smaller that the WTM the operation is rejected and the transaction that has requested the operation is killed.

  $$ts < WTM(x) \rightarrow \text{rejected}$$

  Intuitively a read is rejected if it wants to be executed after a transaction with a bigger timestamp has already written the same data. In other words **the read operation is rejected when it's late with respect to write operations**.

- otherwise the operation is granted and the value of RTM is updated with the maximum between RTM and ts

  $$ts \geq WTM(x) \rightarrow \text{accepted and } RTM(x) = \max(RTM(x), ts)$$

Similarly if when a transaction requests a write operation $w_{ts}(x)$

- If the timestamp of the request is smaller that the WTM or than the RTM the operation is rejected and the transaction that has requested the operation is killed.

  $$ts < WTM(x) \vee ts < RTM(x) \rightarrow \text{rejected}$$

  Intuitively **a write operation is rejected when it's late with respect to read and write operations**.

- otherwise the operation is granted and the value of WTM(x) is set to ts.

  $$ts \geq WTM(x) \wedge ts \geq RTM(x) \rightarrow \text{accepted and } WTM(x) = ts$$

A summary of the conditions above is shown in Table 2.5.

	Rejected when	Accepted when
Read $r_{ts}(x)$	$ts < WTM(x)$	$ts \geq WTM(x)$ $RTM(x) = \max(RTM(x), ts)$
Write $r_{ts}(x)$	$ts < WTM(x) \vee ts < RTM(x)$	$ts \geq WTM(x) \wedge ts \geq RTM(x)$ $WTM(x) = ts$

Table 2.5: Operation rejection or acceptance conditions

Dirty reads

The basic version of timestamp control assumes that all transactions commit, thus it allows dirty read anomalies. To solve dirty reads we have to introduce a condition on the operations. In particular a transaction T_t that issues a $w_t(x)$ or a $r_t(x)$ such that $t > WTM(x)$ (i.e. acceptable) has its read or write operation delayed until the transaction T' that wrote the values of x has committed or aborted. In other words we are delaying the operation that read or write an object already written by another transaction.

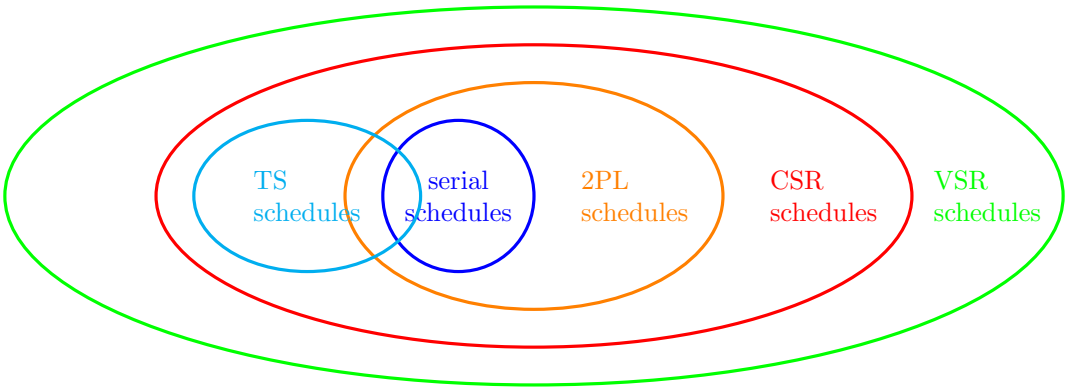

Figure 2.8: TS class.

2.5.2 Timestamp equivalence

The class of schedules that are accepted by the timestamp method is called TS. Let's confront TS with $2PL$, CSR and VSR.

There exist schedules that are in TS but not in $2PL$. For instance the schedule

$$r_1(x)w_1(x)r_2(x)w_2(x)r_0(y)w_1(y)$$

is accepted by the timestamp method but not by the 2 phase locking (the transaction 1 has to release a lock on $w_1(x)$ and then take it again to execute $w_1(y)$, which is not allowed).

On the other hand there are schedules that are in $2PL$ but not in TS. An example is the schedule

$$r_2(x)w_2(x)r_1(x)w_1(x)$$

that is serial (thus view serialisable, conflict serialisable and in $2PL$) but is rejected by the timestamps method because the transactions are executed from the youngest to the oldest.

We can also find schedules that are both in TS and in $2PL$ like

$$r_1(x)r_2(y)w_2(y)w_1(x)r_2(x)w_2(x)$$

In other words the class TS intersects with $2PL$ but doesn't include all serialisable schedules.

Deadlocks When the timestamps method detects an error it kills the transaction. This action doesn't only prevent collisions but also deadlocks. On the other hand the $2PL$ technique puts the transaction in a wait status potentially causing deadlocks.

2.5.3 Thomas rule

As we have seen, timestamp schedulers may kill many transactions. To solve this problem we can change the behaviour for write operations. In particular when a transaction wants to execute a write operation $w_t(x)$

- If $ts < RTM(x)$ the request is rejected and the transaction is killed.
- If if $ts < WTM(x)$ then the write is obsolete, thus it can be skipped.
- Otherwise access is granted and $WTM(x)$ is set to ts.

The updated behaviour is shown in Table

	Rejected when	Skipped when	Accepted when
Read $r_{ts}(x)$	$ts < WTM(x)$		$ts \geq WTM(x)$ $RTM(x) = \max(RTM(x), ts)$
Write $r_{ts}(x)$	$ts < RTM(x)$	$ts < WTM(x)$	$ts \geq WTM(x) \wedge ts \geq RTM(x)$ $WTM(x) = ts$

Table 2.6: Operation rejection or acceptance conditions with Thomas rule.

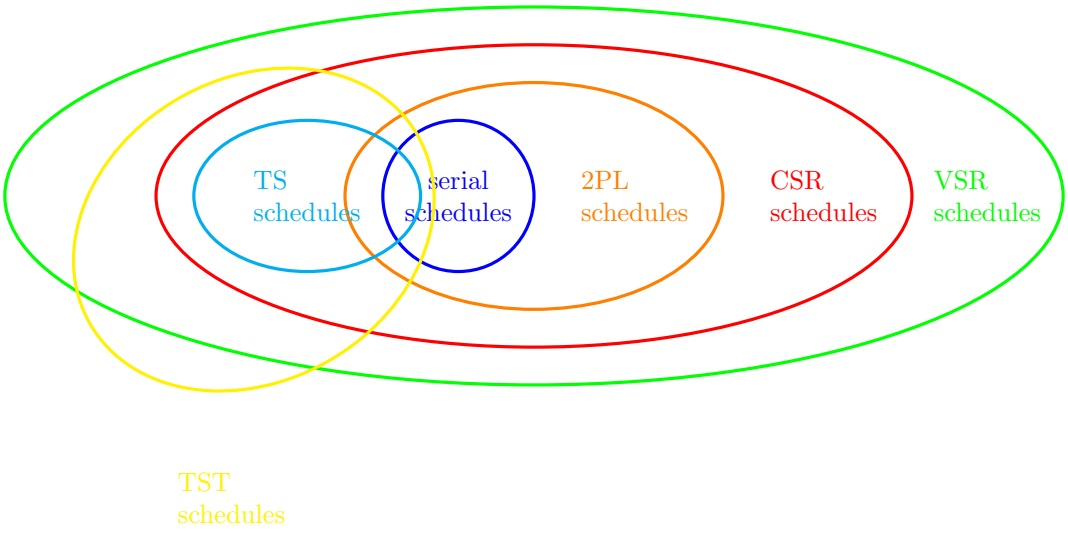

Figure 2.9: TS with Thomas rule class.

2.5.4 Multiversion timestamps

Multiversion extends the timestamps method. In particular every time a write operation occurs (and it's accepted) the system generates a new version of the written resource. Having different versions of the written resource it is always possible to read. To allow multiple read operation there must exist multiple WTM, one for each new copy (but only one RTM for object is required).

In particular

- Read operations $r_{ts}(x)$ are always accepted because they can always read the correct version. To ensure that the correct version is read
  - If the timestamp of the reading transaction ts is greater than a $WTM_N(x)$ then the copy related to such WTM is returned.
  - Otherwise we have to take the copy x_k so that $WTM_k(x) \leq ts < WTM_{k+1}(x)$.

  As for TS without multiversion, the value of $RTM(x)$ is updated with the maximum between the current value and the timestamp of the read operation.

$$RTM(x) = \max(RTM(x), ts)$$

- Write operations

- Are rejected if the timestamp ts of the writing transaction is smaller than the timestamp $RTM(x)$. Such writes are rejected because a younger transaction is trying to write a value previously read by an older transaction.
- Are accepted and a new version of the resource is created. Also a new timestamp WTM is created with value ts

  $$WTM_N(x) = ts$$

  N is incremented every time a new timestamp is created (e.g. starts from 1 and increments to 2, 3, 4, ...).

A summary of rejection and acceptance conditions for write and read operations in TS-multi is shown in Table 2.7.

	Rejected when	Accepted when
Read $r_{ts}(x)$	never	$ts \geq WTM_N(x) \Rightarrow r(x_N)$ else $r(x_k) : WTM_k(x) <= ts < WTM_{k+1}(x)$ $RTM(x) = \max(RTM(x), ts)$
Write $r_{ts}(x)$	$ts < RTM(x)$	$ts \geq RTM(x)$ $WTM_N(x) = ts$

Table 2.7: Operation rejection or acceptance conditions with multiple timestamps.

Snapshot isolation

Multiversion timestamps can be handled in DBMS with a specific isolation level. In particular the `SNAPSHOT ISOLATION` specifies that only WTMs are used (no RTMs). When setting the snapshot isolation level, a transaction T

- Reads the version consistent with its timestamp (i.e. the version that existed when the transaction T started). The version that existed when the transaction started is called **snapshot**.
- Defers all the write operation to the end.

When the scheduler detects that the writes of a transaction conflict with writes of other concurrent transactions after the snapshot timestamp, it aborts.

Snapshot isolation does not guarantee serialisability. Let us consider the following example

1. T1: `update Balls set Color=White where Color=Black`.
2. T2: `update Balls set Color=Black where Color=White`.

In this case serialisable executions of T1 and T2 will produce a final configuration with balls that are either all white or all black (depending on the fact that T1 is executed before T2 or vice-versa). An execution under Snapshot Isolation in which the two transactions start with the same snapshot will just swap the two colours. This anomaly is called **write skew**.

Part II

Java Persistance Api

Chapter 3

Java Enterprise Edition

Java Enterprise Edition defines a set of guidelines that specify how an app should be build, in fact Java EE offers a set of classes and interfaces that allow to write critical parts of an application in a simple and clean way.

Among all the utilities that Java EE offers we can find a set of interfaces that can be used to write a Web application connected to a database.

Characteristics Java EE

- Offers a set of **components** that can be assembled to create complex applications. A component is a Java class that follow the guidelines defined by Java EE.
- Offers a set of **containers** that allow to connect the components to the lower layer functionalities offered by the platform on which the app is developed. Containers are used to divide the functionalities of the components. Every container has a specific location on the multi-tier structure of a Web-App. For instance the application client container handles the application client and the web browser and has to be localted on the client machine.
- Is declarative (i.e. we specify a property and we don't have to do any implementation).

3.1 Java DataBase Connectivity

Java DataBase Connectivity (JDBC) is an API that allows to

- **Connect** to a database.
- **Write** the database.
- **Read** the database.

In particular we have to

1. Open a connection to a database.
2. Use standard SQL queries to read and write the database. Such queries are written using Java's programming language methods (i.e. we don't have to explicitly write SQL queries).

3.2 Servlet

Servlets are components used for Web applications to control the interactions between the user and the server. In particular servlets offer method to

- Retrieve information from the client.
- Handle the information of the client.

In other words the servlet classes allows to implement an HTTP, request-response web server.

Container A servlet is executed inside a **servlet container** that offers additional services like concurrency and lifecycle management.

3.3 Enterprise Java Beans

Enterprise Java Bean is the component on the server side that handles the business logic of the app.

Container EJB components execute in a EJB container that offers services for

- Lifecycle management.
- Transaction management.
- Replication.
- Scaling.

3.4 Java Persistence API

The Java Persistence API is the specification of an interface for mapping relational data (i.e. tuples in a database) to object oriented data in Java (i.e. Java objects).

3.5 Java Transaction API

The Java Transaction API is used for handling transactions. In particular it allows component to start, commit and rollback a transaction.

Transactions can be expressed using

- Programmatically using the methods offered by the `UserTransaction` interface.
- Using Java annotations.

Part III

Ranking queries

Chapter 4

Introduction

4.1 Multi-object optimisation

Data in a database is always represented using multiple attributes. The objective of multi-object optimisation is, given N objects described by d attributes, to find the best k objects. In particular we would like to optimise multiple criteria.

For instance consider a dataset of processors described by `reliability`, `price` and `performance`. Our goal is to find the processors that with the best performance but the minimum cost (we want to maximise performance and minimise cost).

Possible usages Multi-object optimisation problems can be found in a variety of fields including

- **Web search**, in fact a search engine has to get the best web pages that correspond to what the user searched.
- **E-commerce websites** in which products have to be filtered using multiple criteria (e.g. a user may want a cheap used car with few kilometres).
- **Recommended systems** that have to suggest a product using the product already viewed by the user.
- **Caches** that have to keep the best (usually the most used) objects.
- **Machine learning**, in particular k-neighbours algorithms have to find, given N points in a d-dimensional space and a point q, the k closest points to q.

Approaches Multi-object optimisation can be approached using

- **Ranking queries** (i.e. top-k queries) that rank an object using a set of criteria.
- **Skyline queries** that return the k objects that are best for some users and not for just a user (like in top-k ranking).

Rank	User 1	User 2	User 3
1^{st}	Audi	BMW	Audi
2^{nd}	Volkswagen	Audi	BMW
3^{rd}	BMW	Volkswagen	Volkswagen

Table 4.1: Ranking of three car manufacturers.

4.2 Historical solutions

Rank aggregation The ranking queries' problem can be seen as a voting problem. In particular, every voter ranks the candidates (i.e. the objects), then all the votes have to be collected and a unique ranking has to be build. In other words we want to combine different ranking lists of objects to produce a single consensus ranking. Every ranking list can use a different attribute to rank its object and then all the lists (that use different attributes for ranking) are combined in a robust way to create a unique ranking that takes in consideration all the attributes.

The ranking aggregation problem has been tackled using different approaches.

4.2.1 Penalty

The penalty approach uses penalties to rank queries. In particular the object in position n gets n penalty points (this means that the top object has the less amount of penalty points). To show how the penalty system works let us consider Table 4.1 in which three users have rated three car brands (*Audi*, *BMW* and *Volkswagen*).

Borda winner

The winner, according to Borda's proposal, should be the one that has obtained the less penalty points considering all the lists. In particular to build the single ranking list we have to

1. Sum the penalty points for each object considering each list. For instance if p_o^l is the number of penalty points obtained by object o in list l then the total penalty of o is

$$p_o^{tot} = \sum_{l \in L} p_o^l$$

2. Order the object with summed penalties in increasing order so that the object with the smallest total amount of penalty points is the winner.

If we consider the car manufacturer's example (Table 4.1)

- Audi got 1 (from user 1) + 2 (from user 2) + 1 (from user 3) = 4 penalty points.
- BMW got 3 (from user 1) + 1 (from user 2) + 2 (from user 3) = 6 penalty points.
- Volkswagen got 2 (from user 1) + 3 (from user 2) + 3 (from user 3) = 8 penalty points.

For this reason the best object is Audi (with the fewest penalty points), followed by BMW (6 points) and Volkswagen (8 points).

Condorcet winner

The winner, according to Condorcet's proposal, should be the one that has won the most duels with every other candidate.

In other words for every couple of objects (i, j) we have to count in all lists how many times i ranked higher than j. To represent the ranking we can use a directed graph in which

- Every node represents an object.
- An arc from node i to node j represents that i won more duels with j.

Example For instance if we consider the car manufacturers' rankings in Table 4.1 we obtain the graph in Figure 4.1, in which

- Audi is connected to BMW and Volkswagen since it wins against both of them (Audi is higher than BMW for user 1 and 3 and higher then VW for every user).
- BMW is connected to Volkswagen since it wins only against VW (BMW ranks higher in user 2 and 3's preferences).

Analysing the graph (Figure 4.1) we can say that

- Audi is the overall winner because it wins over BMW and Volkswagen. In other words Audi has no incoming arcs.
- BMW is second because it wins over Volkswagen. Graphically, after removing the winning node, BMW has no incoming arcs.
- Volkswagen is last because it has no outgoing arcs (i.e. it hasn't won any duel).

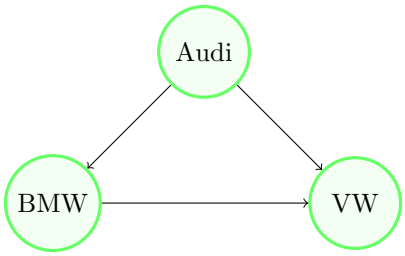

Figure 4.1: The graph to represents Condorcet ranking according to voting in table 4.1

Cycles Both Borda's and Condorcet's proposals can lead to cyclic graphs meaning that a it's not possible to find a winner. For instance the votes in table 4.2 generate a cyclic graph so it's not possible to define a winner using both Borda's and Condorcet's proposal.

4.2.2 Arrow's axiomatic approach

Arrows defined that no rank-order electoral system can satisfy the following criteria

- No dictatorship is present.

Rank	User 1	User 2	User 3
1^{st}	Audi	BMW	Volkswagen
2^{nd}	Volkswagen	Audi	BMW
3^{rd}	BMW	Volkswagen	Audi

Table 4.2: Ranking votes that generate a cyclic graph using Breda's or Condorcet's method.

- If all prefer object X to Y, then the group prefers X to Y.
- If, for all voters, the preference between X and Y is unchanged, then the group preference between X and Y is unchanged.

4.2.3 Metric

The metric approach to ranking is based on distances between rankings. In particular we want to find a ranking R that minimises the total distance to the rankings $R_1, \ldots, R_d$.

The distance between two rankings R_1 and R_2 can be defined in many ways but we will analyse two examples

- Kendall's tau distance $K(R_1, R_2)$.
- Spearman's footrule distance $F(R_1, R_2)$.

Kendall's tau distance

Kendall's tau distance $K(R_1, R_2)$ between two rankings R_1 and R_2 is defined as the number of swaps we have to do to convert R_1 in R_2.

If we consider the car's manufacturers example (Table 4.1) the distance between user 1's ranking and user 2's ranking is 2 since we have to swap Volkswagen and BMW and then swap BMW and Audi.

Complexity Computing Kendall's tau distance is hard, in fact it's a **NP-complete** problem.

Spearman's footrule distance

Spearman's footrule distance $F(R_1, R_2)$ between two rankings R_1 and R_2 adds up the distance between the ranks of the same object in the two rankings.

$$F(R_1, R_2) = \sum_{o \in O} |r_{o,1} - r_{o,2}|$$

Let us consider the car manufacturer (Table 4.1) once again. The Spearman's footrule distance between the ranking of users 1 and 2 is

$$d_{12} = |r_{Audi,1} - r_{Audi,2}| + |r_{BMW,1} - r_{BMW,2}| + |r_{VW,1} - r_{VW,2}| = |1-2| + |3-1| + |2-3| = 4$$

Complexity Spearman's footrule distance $F(R_1, R_2)$ can be computed in polynomial time, thus it's easier than computing Kendall's tau distance.

Approximation Kendall's tau distance K approximates Spearman's footrule distance F, in fact F is between K and $2K$.

$$K \leq F \leq 2K$$

As we can see, at least from our example, the rule is confirmed.

$$2 \leq 4 \leq 2 \cdot 2$$

4.3 Opaque rankings

Opaque rankings use only the position of an object in the ranking lists to build a single ranking. This means that we don't assign a score to every object.

4.3.1 MedRank

Let us consider m ranked lists $R_1, \ldots, R_m$ of N elements each. We want to obtain the top k elements according to median ranking. In particular we have to

1. Use sorted access in each list in parallel, one element at a time, until there are k elements that occur in more than $m/2$ lists.
2. The k elements that appear in more than $m/2$ lists are the top k elements.

Optimality MedRank is not optimal, in fact we can find some algorithm that can beat MedRank. MedRank is **instance optimal**, in fact there is no algorithm that can beat MedRank by some fixed factor (that for MR is 2). This means that there is no algorithm that can beat MedRank with more than twice the number of accesses to the lists.

Example To better understand MedRank let us consider a website that has to rank hotels with respect to three characteristics (e.g. `Price`, `Stars` and `Breakfast`). The hotels are ranked with respect to such characteristics in Table 4.3 and the result is shown in Table 4.4. The median between the ranks defines the order of the top k elements.

Rank	Price	Stars	Breakfast
1^{st}	**Hilton**	Barrière	*Marriot*
2^{nd}	Intercontinental	Wyndham	Accor
3^{rd}	Best Western	**Hilton**	Hyatt
4^{th}	Aman Resorts	Eurohotel	AirBnB
5^{th}	Eurohotel	*Marriot*	**Hilton**

Table 4.3: Ranking queries for the hotel example.

hotel	median rank
Hilton	$median(\{1, 3, 5\}) = 3$
Marriot	$median(\{1, 5, ?\}) = 5$
Eurohotel	$median(\{4, 5, ?\}) = 5$

Table 4.4: The top 3-query result for queries 4.3.

Chapter 5

Ranking queries

With opaque queries we have only considered the order of each object in different rankings. Now we want to introduce a number that represents how good an object in a ranking is. In particular the scoring function $s(t)$ assigns a score to tuple t.

1-1 queries When computing the $s(t)$ function we might have to join multiple tables. For instance if we want to compute the best football player we might have to consider goal scored and assist done and such statistics might be in different tables (e.g. `Goal` and `Assist` tables).

To describe ranking queries we are going to consider queries in which the tables that have to be joined have a 1 on 1 relationship (e.g. the key of `Goal` has one and only one corresponding key in `Assist`). This is the simplest case to deal with but it can also be generalised to a $M - N$ relationship.

Access Tuples in lists can be accessed in two ways

- **Sorted access**. With sorted access a list is passed from start to end accessing one tuple at a time.
- **Random access**. Random access allows to directly obtain (i.e. in constant time) a tuple given its id, independently from the position of the tuple in the list. Random access works like in Random Access Memories.

Score function The score function $s(t)$ should be **monotonic**, this means that the overall score of a tuple increases when a partial score from a list is added. For instance if we read the score of the tuple t_1 in a list a and the we read the score of t_1 in another list b then the overall score $s(t_1^a, t_1^b)$ should be bigger than the score of $s(t_1^a)$

$$s(t_1^a, t_1^b) > s(t_1^a)$$

Normalised scores Usually it's convenient to considered normalised scores, that is scores between 0 and 1. To normalise a list of scores we can

1. Find the lowest S^- and highest S^+ score.
2. Subtract $S^+ - S^-$ to every score.
3. Divide every score by S^+.

5.1 Naive approach

If we assign a score to each tuple t we can

1. Sort the tuples in decreasing order of s.
2. Take the first k tuples.

This simple algorithm can be written in pseudocode as follows

```
foreach t:
    compute s(t)
sort tuples in increasing s(t)
return first k tuples
```

Notice that in this case we haven't created different rankings but we just gave a score to each tuple (i.e. object), in fact the score already considers the different attributes of the tuples. For instance if we are ranking football players the score might be the sum of goals and assists.

```
s(t) = t.goals + t.assists
```

We can also add weights to give more importance to a certain attributes

```
s(t) = t.goals * 0.8 + t.assists * 0.2
```

Problems This solution is very easy but also has some problems. In particular we have to sort the tuples, which is an expensive operation. The computation is even more expensive if we have to join multiple tables to calculate the score function (in our example the goals might be in a table while the assists in another).

5.1.1 Use in SQL

This naive approach needs two operations

- Sort the tuples.
- Get a limited (k) number of tuples.

Sorting SQL allows us to sort tuples using the `ORDER BY` syntax.

Obtaining a limited number of tuples Since 2008 SQL allows to limit the number of tuples using the syntax

```
FETCH FIRST <k> ROWS ONLY
```

The same result can be obtained in some dialect using non-standard syntax, in particular

- Oracle allows the syntax

  ```
  ROWNUM <= k
  ```

- PostgreSQL and MySQL allow the syntax

```
LIMIT k
```

- Microsoft SQL Server allows the syntax

```
SELECT TOP k FROM
```

For instance query 5.1 is a valid top-10 query in MySQL.

```
SELECT *
FROM Stats S
ORDER BY S.Goal + S.Assist
LIMIT 10
```

Listing 5.1: A valid top-10 query.

If we consider that the stats are in two different tables `OffenceStats (Player, Goals)` and `MidfieldStats (Player, Assists)` then query 5.2 can be used to obtain the top-10 players (giving more importance to goals)

```
SELECT *
FROM OffenceStats O JOIN MidfieldStats M
ON O.Player = M.Player
ORDER BY 0.7 * O.Goals + 0.3 * M.Assists
LIMIT 10
```

Listing 5.2: A valid top-10 join query.

Non deterministic semantics If multiple tuples have the same score than the query may return different sets of query. Such sets are all valid and satisfy the constraint of the query.

5.2 Geometric representation

To better understand how ranking queries work let us explore their graphical representation. To do so we will consider two characteristics of an house: `NumberOfBedrooms` and `GardenDimensions`. The houses are ranked in Table 5.1 and their representation is shown in Figure 5.1. Let us consider the following scoring function

```
s(t) = 0.2 * t.NumberOfBedrooms + 0.8 * t.GardenDimensions
```

If we fix a constant value v for $s(t)$ we obtain a line that has equation

$$v = 0.2b + 0.8d$$

or

$$b = -\frac{0.8}{0.2}d + \frac{1}{0.2}v$$

These lines (one for each value of v we choose) represent all the points with the same scoring value. If we consider our example we can notice that hotel 1 and 2 lie on the same line $b = -4d + 5$, thus their score is in both cases

$$b + 4d = 5$$
$$2b + 8d = 10$$
$$0.2b + 0.8d = 1 = s(t)$$

	Garden Dimensions
house 3	4000
house 4	4000
house 5	2000
house 2	1000
house 1	0

	Number Of Bedrooms
house 5	5
house 1	5
house 3	4
house 4	3
house 2	1

Table 5.1: Houses ranked by garden dimensions (in m^2) and number of bedrooms.

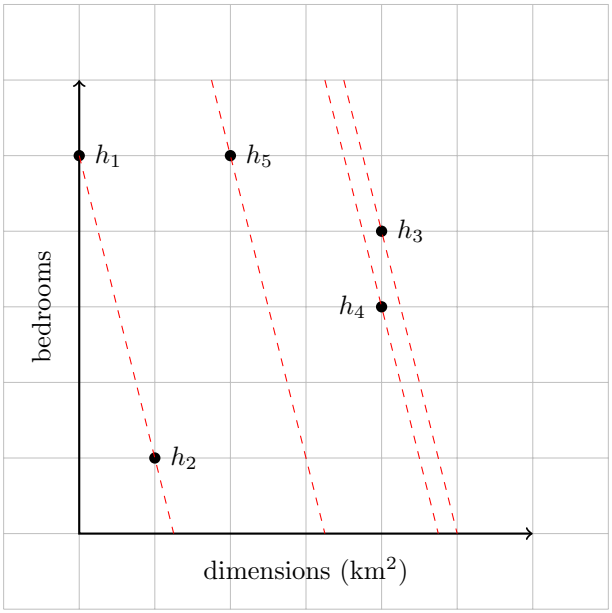

Figure 5.1: Graphical representation of Table 5.1.

Changing the weight of the attributes in the scoring function means changing the lines with equal score.

Goal The goal of ranking queries is to minimise or maximise the scoring function. This goal can be represented as a point in the space, in fact if we want to minimise the scoring function we might set the goal as the origin (i.e. (0,0)).

5.2.1 K-nearest neighbours

In some cases we might want to obtain the results that are closer to some ideal object without explicitly minimise or maximise the scoring function. Consider for instance the house example. In this case we might want to obtain the k-houses that are closer to an ideal house with 3 bedrooms and a 3000 m^2 garden (i.e. the blue square point in Figure 5.3). In this case it's not possible to express the problem like a minimisation or maximisation of a scoring function. On the other hand we can formulate the problem as a distance problem, in fact we want to find the houses that are closest (in terms of number of bedrooms and garden dimensions) to the ideal house.

Distance function

The main problem in the k-nearest problem is computing the distance between two points, in fact for every point p we have to compute the distance between p and the goal and then return the points with the smallest distance. There exists many functions to compute the distance between two points but we are going to analyse

- The **Minkowski** distance.
- The **Euclidean** distance.
- The **Manhattan** (or city-block) distance.
- The **Chebyshev** (max) distance.

Minkowski distance The Minkowski distance is the most general case of distance and depends on a parameter p. Depending on the value of p we can obtain the other distances (for instance for $p = 2$ we obtain the Euclidean distance). Practically the Minkowski distance $L_p(t, q)$ between two points t and q can be computed as

$$L_p(t,q) = \left(\sum_{i=1}^{m} |t_i - q_i|^p \right)^{\frac{1}{p}}$$

Euclidean distance The Euclidean distance is obtained from the Minkowski distance setting $p = 2$.

$$L_2(t,q) = \sqrt{\sum_{i=1}^{m} |t_i - q_i|^2}$$

Graphically (in two dimensions) the points at the same Euclidean distance from the goal are on the same circumference.

Manhattan distance The Manhattan distance is obtained from the Minkowski distance setting $p = 1$.

$$L_1(t,q) = \sum_{i=1}^{m} |t_i - q_i|$$

Graphically (in two dimensions) the points at the same Manhattan distance from the goal are on the same rhombus.

Chebyshev distance The Chebyshev distance is obtained from the Minkowski distance setting $p = \infty$.

$$L_\infty(t,q) = \max_i |t_i - q_i|$$

Graphically (in two dimensions) the points at the same Chebyshev distance from the goal are on the same square.

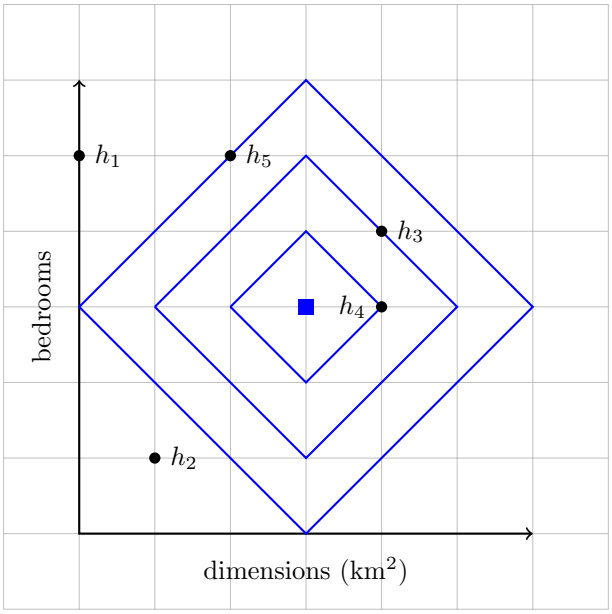

Figure 5.2: Manhattan distance.

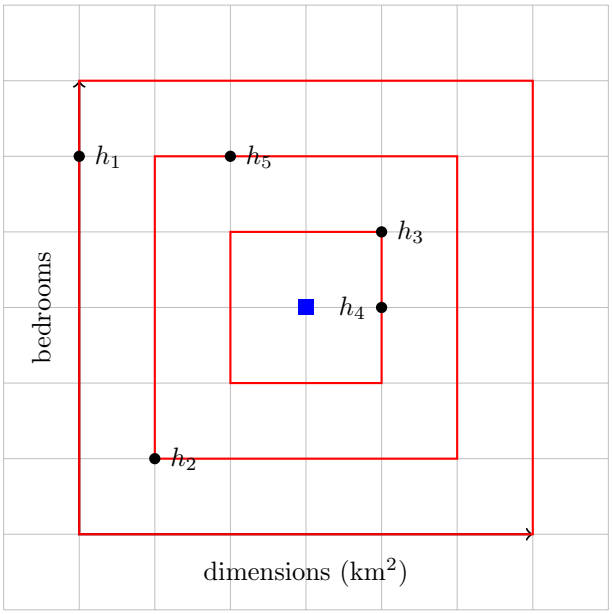

Figure 5.3: Chebyshev distance.

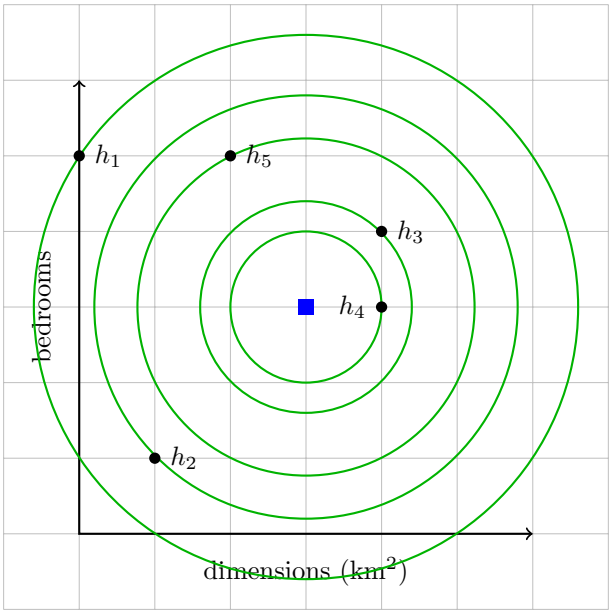

Figure 5.4: Euclidean distance.

5.3 B-zero

The B-zero algorithm uses max as scoring function.

$$s(t) = \max(s(t)) = \max s(t_1), ..., s(t_n)$$

Algorithm 1 (B zero)**.** *The B_0 algorithm works as follows*

1. *Make k sorted accesses on each list and store objects and partial scores of the objects in a buffer B.*
2. *For each object in B compute the maximum of its available partial scores.*
3. *Return the k objects with the maximum score.*

Max function It's important to underline that after k accesses we can return the top-k tuples, even if for some objects it's possible to compute only the partial score. This happens only if we use the max function as scoring function because even if for an object t we haven't obtained a score, we can be sure that this score is less than the maximum score at level k (i.e. the last level visited after the algorithm stopped), thus it doesn't have an impact on the result of the max function.

Example To better understand why this algorithm works with the max function we can consider example 5.2. Say that we want to obtain the top-2 queries ($k = 2$), thus we have to access the first two lines. After accessing both lines we obtain the following buffer

```
B = [
        t7 = {0.7, 1.0},
```

```
    t3 = {0.65, 0.6},
    t2 = {0.9, 0.8}
]
```

If we compute the max function for every set in the buffer we obtain that the top-2 tuples are `t7` and `t2` (max values are 1.0 and 0.9 respectively). As we can see the max value for t_2 is 0.9 and we can't find another value for t_2 under level 2 with a bigger value of $\max(t_2)$ because all values under level 2 are smaller than the maximum value of level 2 (which in this instance is 0.8). In other words we can't find a value that is bigger than the k-th value in the top-k because otherwise such value would have been the k-th top value. The same isn't true if we use other scoring functions like sum or min.

ID	score for attribute 1
t_7	0.7
t_3	0.65
t_4	0.6
t_2	0.5

ID	score for attribute 2
t_2	0.9
t_3	0.6
t_7	0.4
t_4	0.2

ID	score for attribute 3
t_7	1.0
t_2	0.8
t_4	0.75
t_3	0.7

Table 5.2: Three ranking lists ranked using different attributes

Access As we can see B_0 only needs sorting accesses, thus no random access is executed.

5.4 Fagin's algorithm

5.4.1 Algorithm

Algorithm 2 (Fagin's algorithm)**.** *Fagin's algorithm*

1. *Extracts the same number of objects using sorted access so that k objects appear in all lists.*
2. *For each object o computes the score $s(o)$. The score might be partial because an object might not have been seen in all lists yet. To get the missing scores and compute the full score we can use random access, in fact we already know the identifier of the object (we have accessed the object at least once).*
3. *Returns the first k objects with highest total score.*

Optimality This algorithm is not instance optimal, this means that we can find another algorithm that in some cases can beat Fagin's algorithm by some arbitrary large (i.e. it can return the result using less accesses).

Complexity Fagin's algorithm has sublinear complexity, in particular it's time complexity is

$$\mathcal{O}(N^{\frac{m-1}{m}} k^{\frac{1}{m}})$$

where

- k is the number of objects to return.
- N is the number of objects in the dataset.
- m is the number of attributes.

Stopping criteria The stopping criteria (i.e. the condition that allows the algorithm to stop) is independent from the scoring function, in fact the algorithm stops when k objects appear in all lists but the score of such objects isn't used to decide when to stop sorted access. This is a bad thing because there is some information (i.e. the score) that we don't fully exploit.

Immediate random access This algorithm can be improved using random access as soon as a row of the lists is visited to immediately obtain all the scores in the other lists. For instance if we visit level 1 of three lists and we find objects t_3, t_7 and t_1, then we access tuple t_3 also in the second and third lists using random access (the same is repeated with t_7 and t_1). This modification improves performance but not by much.

5.4.2 Example

Consider the following example to better understand how Fagin's algorithm works. Let us consider the data-set in Table 5.3 containing hotels in Paris ranked by cheapness and rating. The goal is to find the top-2 hotels ($k = 2$) using the scoring function

$$s(t) = 0.5 \cdot t.cheapness + 0.5 \cdot t.price$$

Hotel	Cheapness
Ibis	0.92
Etap	0.91
Novotel	0.85
Mercure	0.85
Hilton	0.825
Sheraton	0.8
Crillon	0.75

Hotel	Rating
Crillon	0.9
Novotel	0.9
Sheraton	0.8
Hilton	0.7
Ibis	0.7
Ritz	0.7
Lutetia	0.6

Table 5.3: Hotels in Paris ranked by cheapness and rating.

As the algorithm imposes, we access all objects until level 5 when we find out that Novotel, Hilton and Ibis appear in both lists. We stop because we found 2 (actually three but 2 of the three hotel are at the same level) objects that are in both lists.

Now we have to access the score of the hotels we have only partially visited using random access, in particular we visit

- Etap in list 2.
- Mercure in list 2.
- Crillon in list 1.
- Sheraton in list 1.

Now that we have all scores we can compute the overall scores of all the hotels and return the top-2 tuples with the highest overall score. In this case the objects with the highest overall score are

1. Novotel with a score of $s(t_{Novotel}) = 0.5 \cdot 0.85 + 0.5 \cdot 0.9 = 0.875$.
2. Crillon with a score of $s(t_{Crillon}) = 0.5 \cdot 0.75 + 0.5 \cdot 0.9 = 0.825$.

5.5 Threshold algorithm

Threshold algorithm is based on Fagin's algorithm but it also considers the objects' score.

5.5.1 Algorithm

Threshold algorithm works this way

Algorithm 3 (Threshold algorithm)**.**

1. *Do sorted access in parallel in all lists.*
2. *Consider level l. For all objects on level l do random access in the other lists to get the total score.*
3. *Compute the overall score $S(s_1, ..., s_m)$ for object o, if the value is among the top k objects' values seen so far, keep object o and remove the object with the lowest score if the buffer is full.*
4. *Define the threshold point as the last objects seen (i.e. the objects of the last row visited) and the threshold as the score of the threshold point (i.e. the score considering the objects of the last row visited, even if they refer to different tuples).*
5. *If the score of the k-th object is smaller than the threshold repeat from step 1.*
6. *Return the current top-k objects.*

In other words the algorithm visits the objects in the data set using serial and random access until it finds k objects that have a better score than the threshold.

Buffer This algorithm keeps an object only if its overall score is better than one k best score and removes the lowest scoring object if the buffer is full, thus the buffer only keeps k objects. This means that the threshold algorithm needs, in general, a smaller buffer with respect to Fagin's algorithm.

Optimality The threshold algorithm is instance optimal.

5.5.2 Cost

Consider the following parameters to evaluate the cost of this algorithm

- SA, the total number of sorted accesses.
- RA, the total number of random accesses.
- c_{SA}, the base cost of a sorted access.
- c_{RA}, the base cost of a random access.

Thus the total cost of the algorithm is

$$c_{tot} = RA \cdot c_{RA} + SA \cdot c_{SA}$$

General case In general we can consider the cost an equal cost for random and sorted accesses.

$$c_{SA} = c_{RA}$$

Web sources Usually for web sources random access is costly

$$c_{RA} > c_{SA}$$

or even impossible

$$c_{RA} = \infty$$

Limited sorted access In other cases sorted access is more costly

$$c_{SA} > c_{RA}$$

or even impossible

$$c_{SA} = \infty$$

This means that we have to choose the best algorithm depending on the field of application.

5.5.3 Example

It's useful to see an example to better understand how the threshold algorithm works. In particular we will consider the same data-set of Fagin's algorithm example (the data set is shown in Table 5.4 for simplicity). The goal is the same as before, we have to find the top-2 hotels using the threshold algorithm and the average between cheapness and rating as scoring function.

Hotel	Cheapness
Ibis	0.92
Etap	0.91
Novotel	0.85
Mercure	0.85
Hilton	0.825
Sheraton	0.8
Crillon	0.75

Hotel	Rating
Crillon	0.9
Novotel	0.9
Sheraton	0.8
Hilton	0.7
Ibis	0.7
Ritz	0.7
Lutetia	0.6

Table 5.4: Hotels in Paris ranked by cheapness and rating.

To find the top-2 queries we have to execute multiple iterations

1. The first row is accessed using sorted access. The value of t_{Ibis} and $t_{Crillon}$ are retrieved using random access in table 2 and 1 respectively. Both hotels are added to the buffer B with their scores.

```
B = {
    'Crillon' : 0.825,
    'Ibis' : 0.81
}
```

The threshold point is the last accessed row

$$\tau = (0.92, 0.9)$$

thus the score of τ is

$$T = s(\tau) = 0.5 \cdot 0.92 + 0.5 \cdot 0.9 = 0.91$$

Because the second element in the buffer is smaller than the threshold score, we have to continue with the next iteration.

2. The second row is accessed using sorted access. The value of t_{Etap} and $t_{Novotel}$ are retrieved using random access in table 2 and 1, respectively. Only Novotel is added to the buffer because it has an higher score than the elements in the buffer. In particular Novotel is added before Crillon because it has an higher score and Ibis is removed because the buffer contains only the top-2 scores.

```
B = {
    'Novotel' : 0.8.725,
    'Crillon' : 0.825
}
```

The threshold point is the last accessed row

$$\tau = (0.91, 0.9)$$

thus the score of τ is

$$T = s(\tau) = 0.905$$

Because the second element in the buffer is smaller than the threshold score, we have to continue with the next iteration.

3. The third row is accessed using sorted access. The value of $t_{Sheraton}$ is retrieved using random access in table 1 (Novotel has already been considered). Sheraton isn't added to the buffer because it's score is smaller of both scores in the buffer.

```
B = {
    'Novotel' : 0.8.725,
    'Crillon' : 0.825
}
```

The threshold point is the last accessed row

$$\tau = (0.85, 0.8)$$

thus the score of τ is

$$T = s(\tau) = 0.825$$

The second element in the buffer equals (so it's not smaller than) the threshold score, so we return the buffer that contains the top-2 tuples.

5.6 Non Random Access

The Non Random Access (NRA) algorithm doesn't use random accesses thus it can be used when the cost of such accesses is too high or it's even impossible to do random accesses.

5.6.1 Algorithm

Description The algorithm keeps a lower and an upper bound for the score of every object and objects are sorted in decreasing order of lower-bound. For this reason this algorithm doesn't return the precise score for every object in the top-k queries but it only ensures that the top-k objects are returned.

Lower bound The lower bound of object t is computed as the sum of the partial scores of t. For instance if we consider the Tables 5.5 the upper bound of Etap is the sum of 0.9 and 0.6 (i.e. the known scores after accessing the first two rows).

Upper bound The upper bound of tuple t is computed as the sum of the partial scores of t already discovered and the minimum scores of the tables in which t hasn't been found yet. For instance if we consider the Tables 5.5 the upper bound of Etap is the sum of 0.9 and 0.6 (i.e. the know scores after accessing the first two rows) and 0.75 (i.e. the lowest score in the second list, which is the list in which Etap hasn't appeared yet).

Threshold The threshold point is the set of scores of the last row visited. The threshold is the value of the threshold point. In our example (Table 5.5), if we assume we have just visited row 2, the threshold point τ is $\tau = (0.9, 0.75, 0.6)$ and the threshold T is $T = s(\tau) = 0.9 + 0.75 + 0.6 = 2.25$.

Hotel	Cheapness
Ibis	1.0
Etap	0.9
⋮	⋮

Hotel	Rating
Crillon	0.8
Novotel	0.75
⋮	⋮

Hotel	Cheapness
Etap	0.6
Hilton	0.6
⋮	⋮

Table 5.5: Upper and lower bounds for NRA algorithm.

Buffer The algorithm uses a buffer B in which the discovered tuples are ordered in decreasing lower bound values.

Algorithm 4 (Non Random Access)**.** *The NRA algorithm*

1. *Makes sorted accesses to each list in parallel.*
2. *Stores in the buffer B each retrieved object o and maintains the lower score $S^-(o)$, the upper score $S^+(o)$ and the threshold τ.*
3. *Repeats from point 1 until the lower bound of the k-th tuple is bigger than the bigger between the threshold $S(\tau)$ and the maximum upper bound among the elements with indices bigger than k. Consider buffer B as 1-indexed.*

$$s^-(B[k]) \geq \max\{\max\{S^+(B[i]), i > k\}, S(\tau)\}$$

Halting condition Call RES the subset of the buffer that contains the best k tuples. NRA basically stops when it can't find an object o' that can do better than any object o in RES. An object o' can't do better than an object o when the upper-bound of the former is smaller than the lower-bound of the latter.

$$S^+(o') \leq S^-(o)$$

Cost function The cost doesn't grow in a monotonic way with k, in fact in some cases it might take less to find the top-$(k+1)$ objects than the top-k objects.

Optimality The NRA algorithm is instance optimal among the algorithms that only allow sorted access.

Algorithm	Scoring function	Data access	Instance optimal	Exact score
B_0	max	sorted	yes	yes
Fagin's	monotone	sorted, random	no	yes
Threshold	monotone	sorted, random	yes	yes
NRA	monotone	sorted	yes	no

Table 5.6: A summary of all algorithms used in ranking queries.

Chapter 6

Skyline queries

Ranking queries are very effective but also have some problems, in particular we have to specify the scoring function $s(t)$ and eventual weight. Skyline queries solve this problem.

6.1 Skyline

Dominance A tuple t dominates a tuple s if and only if

- t is not greater than s in all dimensions.

$$t[i] \leq s[i] \quad \forall i \in 1, ..., m$$

- t is smaller than s in at least one dimension.

$$\exists j \in 1, ..., m : t[j] < s[j]$$

Considering both conditions we obtain

$$t \prec s \iff \forall i \; t[i] \leq s[i] \wedge \exists i : t[i] < s[i]$$

From this definition we can notice that a small value is a good value, opposite to ranking queries in which a good value is an high value.

Skyline The set of points (i.e. tuples) not dominated by any point is called **skyline**. In other words a tuple t is in the skyline if and only if it's the top-1 result with respect to at least one attribute. The skyline can also be interpreted as the set of potentially optimal tuples considering different scoring functions. An example of skyline is shown in Figure 6.1. Notice that

- Point S_1 is in the skyline since no point has a smaller value for attribute$_2$, thus S_1 isn't dominated by any other point.
- Point S_2 is in the skyline since no point that is smaller or equal to S_2 in all dimensions, in fact
  - S_1 is smaller only with respect to attribute$_2$.
  - S_3 is smaller only with respect to attribute$_1$.

  - (3, 4) is equal with respect to attribute$_1$ but strictly greater with respect to attribute$_2$.

- Point S_3 is in the skyline since no point has a smaller value for attribute$_1$, thus S_3 isn't dominated by any other point.

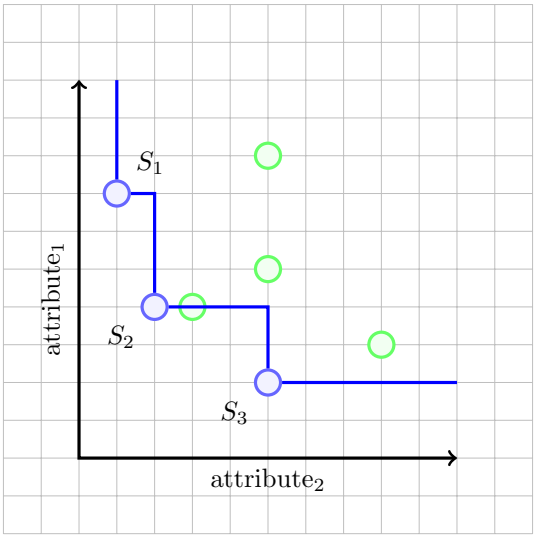

Figure 6.1: A graphical representation of the skyline.

6.1.1 SQL

In SQL we can define the skyline using the following syntax

```
SELECT * FROM Table T WHERE Condition
SKYLINE OF [DISTINCT] T.d1 [ MIN | MAX | DIFF ],
                      T.d2 [ MIN | MAX | DIFF ], ...,
                      T.dm [ MIN | MAX | DIFF ],
```

where

- `d1`, ..., `dm` are the dimensions of the skyline.
- `[ MIN | MAX | DIFF ]` specifies if the dimension has to be minimised, maximised or has to be different.

In general a complete query that computes the skyline can be written as follows

```
SELECT ... FROM ... WHERE ...
GROUP BY ... HAVING ...
SKYLINE OF [DISTINCT] d1 [MIN | MAX | DIFF], ...,
                      dm [MIN | MAX | DIFF]
ORDER BY ...
```

This query can be rewritten using a nested query but it would be very inefficient and slow.

6.1.2 Advantages and disadvantages

Skyline queries give an overview of all dataset, in fact they return a set of all interesting points with respect to different scoring functions. The skyline can be used to choose the best tuple depending on specific criteria.

Correlated and anti-correlated datasets Skyline queries aren't that efficient in case of datasets that are

- Correlated (i.e. points on a line $y = mx + q$), in fact in this case the skyline only contains the closest point to the origin.
- Anti-correlated (i.e. points on a line $y = -mx + q$), in fact in this case the skyline contains all the points of the dataset.

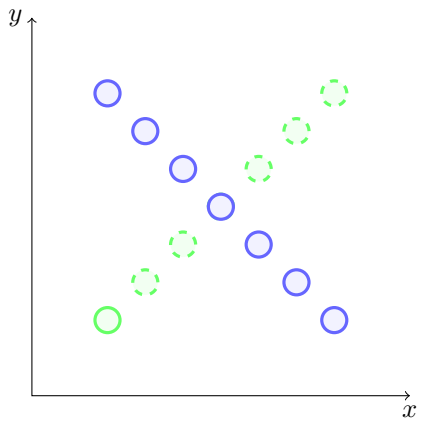

Figure 6.2: Correlated (green) and anti-correlated (blue) datasets.

6.2 Algorithm

6.2.1 Block nested loop

The block nested loop (BNL) algorithm allows to find the skyline without using a nested loop. In particular the algorithm is shown in Listing 6.1. The algorithm receives as input a dataset `D` of multi-dimensional points, and returns the skyline of `D` .

```
1    let W = {}
2    foreach p in D:
3        if p not dominated by any s in W:
4            remove from W the points dominated by p
5            add p to W
6    return W
```

Listing 6.1: Block nested loop alorithm.

The if condition checks if no point in `W` dominates `p`, in fact `p` isn't in the skyline if at least one point in `W` dominates `p`.

Complexity The complexity of this algorithm is quadratic ($n = |D|$ is the number of points)

$$\mathcal{O}(n^2)$$

in fact the algorithm uses two loop

- A for loop at line 2: `foreach p in D`.
- The check at line 3, in fact to check if a point `p` is in the set `W` we have to access all the array.

6.2.2 Sort filter skyline

The sort filter skyline (SFS) algorithm exploits sorting to reduce the number of accesses to array `W`, in fact if the dataset `D` is sorted then a later tuple can't dominate any previous tuple, because if we sort the dataset, then a tuple s surely has some value that is bigger than the previous tuple t, thus s can't dominate t. This is true only if the dataset is sorted (in increasing order) using a monotone function.

The SFS algorithm is shown in listing 6.2. In particular the algorithm receives as input a dataset `D` of multi-dimensional points, and returns the skyline of `D`.

```
1    let S = D sorted by a monotone function of the attributes of D
2    let W = {}
3    foreach p in S:
4        if p not dominated by any s in W:
5            add p to W
6    return W
```

Listing 6.2: Sort filter skyline algorithm.

Complexity Even if sorting improves performance (especially for large data-sets), the sort filter skyline algorithm still has a quadratic complexity

$$\mathcal{O}(n^2)$$

6.3 k-Skybands

Skyline queries can be extended to k-skybands queries. A k-skyband is the set of tuples dominated by less than k tuples.

Properties If $k_1 < k_2$ then all the points in the k_1-skyline are also contained in the k_2-skyline.

Another important property of a k-skyband is that every top-k result set (obtained with a top-k query) is contained in the k-skyband. Namely, the top k-skyband contains all top-k tuples for some monotone function.

Algorithm We can compute the k-skyband using the same algorithms used for the skyline. The only difference is that in this case a point `p` has to be added to the list `W` only if we can't find k or more points in `W` that dominate `p`. In other words `p` is added to `W` if it isn't dominated by k or more points in `W`.

Skyline A skyline is a 1-skyband ($k = 1$) because the skyline is the set of tuples not dominated by any tuple (i.e. the set of tuples dominated by 0 tuples).

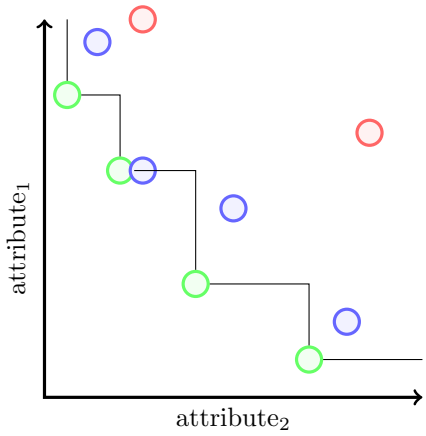

Figure 6.3: Skybands (1-skyband in green, 2-skyband in blue).

Part IV

Physical structure

Chapter 7

Memory structure

Query execution speed is fundamental, especially for large databases. To improve speed we also have to optimise how the tuples are memorised and retrieved.

7.1 Types of memories

A DBMS works with two types of memory

- The **main memory** (e.g. RAM).
- The **secondary memory** (e.g. storage).

Main memory Main memory is very fast but not very big thus it's usually used to execute queries. Main memory is divided in **pages**. This means that the fundamental unit of memory for the main memory is the page.

Secondary memory Secondary memory can hold a lot of data but it's slower than main memory, thus it's used to store tables and tuples. Secondary memory is divided in **blocks**. This means that the fundamental unit of memory for secondary memories is the block.

Size To describe the physical structure of a database we will assume the size of a page equal to the size of a block. This is not always true in practical implementations.

Usually, the size of a page (or a block) is 8 kilobytes.

7.1.1 Input and output

Tuples are stored in secondary memory but executed in main memory, thus we have to move some blocks from secondary memory to main memory and vice versa. Such operations are called input/output (I/O) operations.

Input/output operations are handled

- By the file system in a normal computer.
- Directly by the DBMS in databases. Usually, the DBMS just uses the low-level services offered by the file system, in fact, the data structure is fully controlled by the DBMS to improve performance. In other words, the DBMS controls almost everything in secondary memory.

Cost The cost of input/output operations depends on the technology used to implement the secondary memory. For instance secondary memory can be

- A **Solid State Drive** (SSD). SSDs allow fast I/O operations but are very expensive thus can't be used for very big databases.
- A **Hard Disk Drive** (HDD). HDDs are much slower than SSDs but are very cheap, thus they are still used in huge databases.

7.1.2 Hard disk

A hard disk drive (HDD) is a mechanical storage device made of a stack of rotating disks. Each disk is read and written using a head.

Tracks and sectors Each disk is divided into sectors. The head also defines circumferences (i.e. all the points at the same distance from the centre of the disk) called **tracks**. The intersection between a section and a track defines a block.

The structure of a disk is shown in figure 7.1.

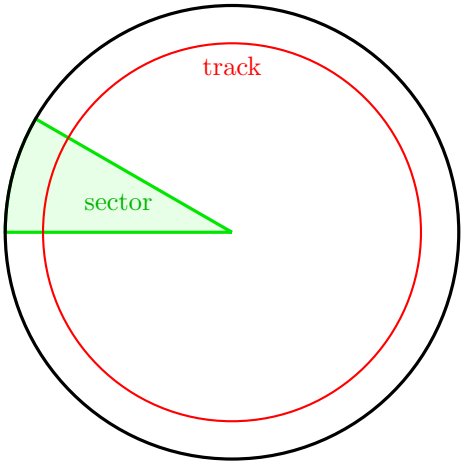

Figure 7.1: Sectors and tracks in a disk.

How to transfer data When an HDD has to transfer some data, it has to execute the following operations

1. **Seek**. The head is moved in the right position (i.e. on the right track). This operation usually takes between 8 and 12 milliseconds.
2. **Latency**. The disk spins until the right sector is reached. This operation usually takes between 2 and 8 milliseconds.
3. **Transfer**. The block of data is transferred to main memory. This operation usually takes around 1 millisecond.

In total, a hard disk needs around 20 milliseconds to transfer a block of data. Main memory is usually 10^4 times faster, thus it's easy to verify that most of the cost of a query is in data transfer. For this reason we can measure the cost of executing a query as the number of I/O operations (i.e. the number of transferred blocks) for every query.

7.2 Block structure

A block contains

- Some **tuples** (in blue) that can also be of variable length (e.g to represents `VARCHAR` attributes). The tuples are implemented as a stack that grows towards the page dictionary.
- A **page dictionary** (in orange) that, for each tuple describes how long the tuple is and its position in the block. In other words the dictionary contains a pointer to the tuples. This dictionary is also implemented using a stack that grows towards the tuples.
- A **checksum** (in pink) to check if the value of the tuples is correct.
- A **block header** and **trailer** (in red) that contain information used by the file system to control the block.
- A **page header** and a **trailer** (in green) that contain information about the access method (more on this in the next chapter).

Figure 7.2: A block of data.

Block factor The block factor B is the number of tuples stored in a block. The block factor can be computed as the ratio between the size of a block S_B and the average size of a tuple S_R

$$\left\lfloor \frac{S_B}{S_R} \right\rfloor$$

Chapter 8

Access methods

The memory structure of secondary memory can be divided in

- **Primary physical data structure**. This structure identifies how tuples are written into blocks of secondary memory. Each table is stored in exactly one primary data structure.
- **Secondary physical data structure**. This structure is an auxiliary structure that allows to speed up access to the blocks. In particular secondary structure allows us to build **indices** on top of blocks to immediately reach one block. A table can have one or many secondary data structures (or even no secondary data structure at all).

Types of access methods Primary and secondary structure can be implemented using three types of access methods

- **Sequential**.
- **Hash-based**.
- **Tree-based**.

Each structure can be implemented using different access methods, in particular

- The primary structure can be implemented using **sequential** and **hash-based** methods but not using the tree-based method.
- The secondary structure can be implemented using **hash-based** and **tree-based** methods but not using the sequential method.

Access method	Primary structure	Secondary structure
Sequential	typically used	not used
Hash-based	sometimes used	frequently used
Tree-based	rarely used	typically used

Table 8.1: A summary of access methods and their usage

8.1 Primary structure access methods

In this section we will analyse how access methods (sequential and hash-based) are used in secondary memory's primary structure.

8.1.1 Sequential

The sequential access method organises data in a sequence of blocks. The memory's header specifies the position of the first block and the next blocks are accessed in sequence (one after the other) using the `next` command that returns the successive block.

Blocks are ordered in memory using one of the following criteria

- **Entry sequence**. When a new tuple is added to memory, it is added to the last block in memory and if needed a new block is created. In other words, the tuples and the blocks are sorted using the entry time. This method is ok for space occupancy and it's efficient to execute queries where all the tuples are queried (`SELECT * FROM Table`) but it's inefficient when the query has to check some condition because all the blocks have to be analysed to verify if such condition is met.
- **Sequence ordered**. In this case, tuples are ordered using a key (i.e. an attribute of the tuple that might also differ from the primary key). Insertion and remove operations are not efficient because we need to sort all the tuples (even if some DBMS implement some advanced techniques to speed up sorting and to avoid global sorting). On the other hand conditional queries can be executed much faster because we can apply binary search and transfer to main memory only the blocks that contain the tuple that satisfy the condition.

8.1.2 Hash-based

The hash-based access method uses a hash table to access tuples in blocks. In particular, a hash table is made of N_B buckets. Each bucket has the same size of a block and contains the tuples with the same hash. The hash function is applied to the key of the tuple, thus when we apply the hash function to a key we obtain the bucket (i.e. the block) in which the tuple is.

Hash function The hash function h is a function that given a key k returns a value between 0 and $N_B - 1$, that is the bucket in the hash table where the tuple is.

$$h : K \to \{0, ..., N_B - 1\}$$

Advantages and disadvantages Hash-based access methods are efficient for small size tables, static content (i.e. tables that don't change much) and equality queries that involve the key (e.g `SELECT a WHERE a.id = 5`) but are not very efficient for interval and full table queries and for tables that change frequently.

Collisions

Sometimes the number of tuples that have the same hash is bigger than the number a block (i.e. a bucket) can contain, thus we have to find a way to handle such exceeding tuples. In particular, we can use

- **Closed hashing**. In this case, the exceeding tuples are inserted in other buckets (usually a successive bucket).
- **Open Hashing**. In this case, a new bucket (overflow bucket) is created and a reference to the overflow bucket is stored in the full bucket. To implement open hashing every bucket needs to have a reference to a list of overflow buckets to store exceeding tuples.

Cost factor

Let us now focus on open hashing. This method creates a list of buckets related to the same key (i.e. hash). This means that when we want to access a tuple using a hash-based primary structure we might have to check in more than one bucket. To account for this extra bucket we have to introduce a cost factor C, which is a factor that has to be multiplied for the number of tuples one wants to access. The cost factor is a function of

- The block factor B.
- The load factor L. The load factor represents the average occupancy of a block and it's computed as the ratio between the occupied slots and the available slots.

$$L = \frac{T}{B \cdot N_B}$$

  where

  - T is the number of tuples.
  - N_B is the number of buckets.
  - B is the number of tuples in a block (hence in a bucket).

For instance, if we want to obtain a tuple with key `id` in a hash table with cost $C = 1.2$ then the total cost of the query is $1 \cdot 1.2 = 1.2$ because to find a tuple we have to access on average 1.2 buckets.

8.2 Secondary structure access methods

8.2.1 Index

An index allows accessing the tuples in a faster way. In particular, an index is implemented using a hash table in which each key is associated with a block.

```
<key, block_pointer>
```

Search keys The keys used in the hash table are made of one or more attributes of a table. It's important to underline that the search key (i.e. the key used in the index hash table) can be different from the primary key of the table, in fact

- The search key can be any attribute or set of attributes.
- The primary key defines constraints (e.g. uniqueness) on one or more attributes.

Types of indices An index can be

- **Sparse** if the table contains only the key of the first tuple in each block.
- **Dense** if the hash table contains all keys.

An index can also be classified in

- **Primary index**. The key used for the index is the same of the one used to organise the tuples in their primary structure.
- **Clustering index**. Multiple blocks can be associated to the same key value.
- **Secondary index**. The index key is different from the key used in the primary structure.

Cost We can define as many indices as we want, but indices are a cost, in fact we have to modify the index structure every time a tuple is inserted. For this reason we have to use indices only when they are really useful, in fact if not used correctly they can even slow down execution.

Here is a list of situations in which indices should or should not be used.

- Don't use indices for small tables.
- Don't add indices if the DBMS automatically creates them (we might add an index already added by the DBMS, and have duplicates).
- Don't use indices when tuples are frequently updated.
- Don't use indices when are executed queries that access a big portion of data.
- Don't use long attributes as an index.
- Use indices if a foreign key is frequently accessed.
- Use indices if queries with `ORDER BY`, `GROUP BY`, `UNION` and `DISTINCT` are frequently executed.

8.2.2 Hash-based

Indices can be directly implemented using hash-tables in which

- The key of the table is the index key.
- The value of the table is the pointer to the block associated with the index key.

Block access To access and transfer to main memory a block we have to

- Access the index of the hash table.
- Access and transfer the actual block pointed by the row in the hash table.

This means that we need 2 total accesses (plus the accesses to eventual overflow buckets in the hash table) to obtain a block.

Advantages and disadvantages Hash-based indices are good when a predicate query (`SELECT a FROM Table WHERE a.id = 5`) is executed but are rather inefficient for other types of queries.

Cost The cost of accessing an index hash table is the same as a primary hash table (also considering overflow buckets). Notice that a secondary memory structure contains indices, thus when we use an index hash table we have to access the blocks of the table to get the pointers and then, if needed, follow the pointers to the primary structure. This means that we have to sum the cost of accessing the primary structure to the cost of accessing the hash table.

8.2.3 Tree-based

Indices can also be implemented using balanced search trees. In particular tree-based indices use B and B^+ trees.

B+ trees

In B^+ trees every node occupies a block and contains multiple keys and values (i.e. a list of key-value couples). The keys are in increasing order.

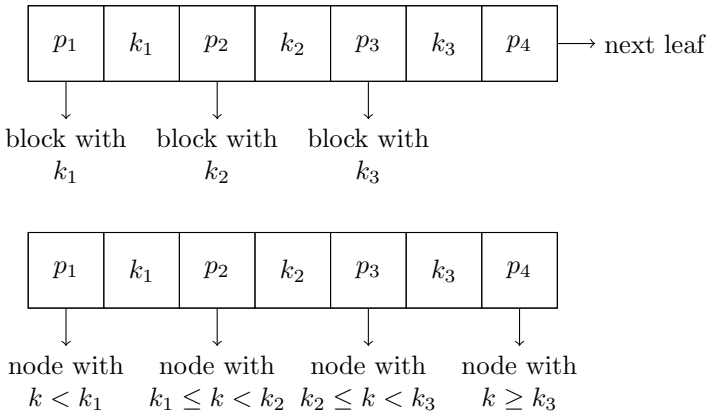

Figure 8.1: An intermediate node (below) and a leaf node (above) in a B^+ tree.

Nodes In a B^+ tree

- **Intermediate** nodes contain pointers to other nodes (intermediate or leaves). Each pointer p_i points to the node with keys between k_i and k_{i+1}. Internal nodes form a sparse index on the leaf nodes.
- **Leaves** contain pointers to the blocks in the main structure, in particular p_i points to the block with key k_i. Leaf nodes are in chained order so that once we reach a leaf we can access the successive leaf in sequential order. For instance if we consider the leaf node in Figure 8.2, p_4 points to the next leaf (i.e. the one in which the first key is k_4). Leaf nodes are a dense index on blocks.

The number of nodes (or leaves) a node points to is called fan-out F. Given the fan-out F and the number of tuples T it's possible to obtain the number of levels L as

$$L = \log_F(T)$$

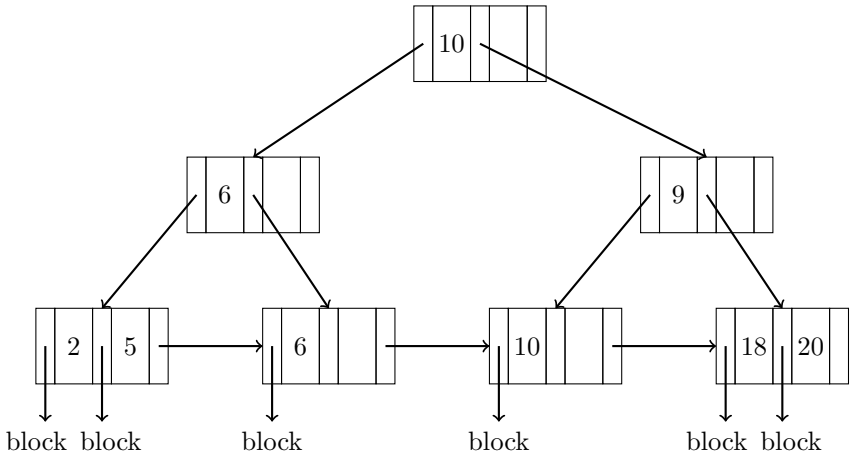

Figure 8.2: A B^+ tree.

Accesses The number of accesses needed to obtain a block is logarithmic with the number of blocks because a B^+ tree is balanced. In other words, the number of accesses is linear with the number of levels of the tree. As for hash-based indices, we have to remember that the leaves contain pointers to the blocks in the main structure, thus if we want to access such tuples in the primary structure we have to sum the access cost of the primary structure to the cost of the B^+ tree.

To improve performance we can also

- Keep the tree shallow (i.e. only a few levels).
- Cache the higher levels of the tree (usually the root).

Another important thing to remember is that the leaves are sorted with respect to the search key, thus, once reached the leaves, we can use sorted access to obtain (in order) the pointers to all the tuples.

B trees

B trees are similar to B^+. The main difference is that B trees eliminate redundant search-keys from intermediate nodes. In particular an intermediate node has, for each key k_i

- One pointer to the sub-tree with keys between k_i and k_{i+1}.
- One (or more) pointer(s) to the block(s) that contain(s) the tuple(s) that have value k_i as key.

In other words, the main difference is that intermediate nodes can either point to another node or directly to some leaves. An example of B tree is shown in Figure 8.3.

Performance Lookup using B trees can be slightly faster, but interval queries are less efficient with respect to B^+ trees.

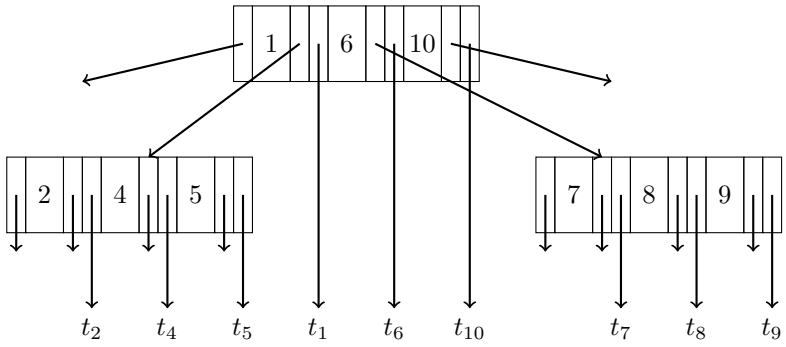

Figure 8.3: A B tree.

8.2.4 Indices in SQL

Indices can be created in SQL using the following syntax

```
CREATE INDEX IndexName ON
TableName(Attribute [, Attribute, ...])
```

Chapter 9

Query optimisation

DBMS usually use a tree to optimise a query. In particular a DBMS

1. Transforms the query in a relational algebra tree.
2. Optimises the tree.

In some cases, the query is even rewritten so that it can be executed more efficiently. For instance, in some cases it's faster to execute a nested query than a join, thus a query with a join is rewritten in an equivalent nested query.

Data dictionary DBMS also use statistics to optimise a query. In particular every table is associated with a data structure, called data dictionary, that stores some statistics about the table. Each data dictionary stores

- The **cardinality** (i.e. the number of tuples) of the table. This field is useful for queries in which the user asks to count how many tuples are present in a table.
- The **maximum** and **minimum** value of each attribute. This field is useful for executing queries with `MAX` and `MIN`.
- The **dimension** in bytes of each attribute.
- The number of **distinct values** of each attribute ($val(Attribute)$). This value is useful to compute the selectivity, i.e. the probability to find a value.

$$selectivity = \frac{cardinality}{val(Attribute)}$$

  For instance in a table `Person(Name, Surname)` if an attribute `Surname` has 20 distinct values and the number of tuples is 100, then we should expect to find on average $100/20 = 5$ people with the same surname in the table (if we assume an homogeneous distribution of surnames).

The data dictionary is updated periodically and can be updated manually using SQL's command

```
UPDATE STATISTICS
```

9.1 Operations

In this section, we will analyse how SQL's main operations are executed and optimised.

9.1.1 Conjunction - AND

Consider a query in which the `WHERE` condition is a conjunction of two predicates. To execute such query it's sufficient to have an index on one of the attributes in conjunction (say the attribute used in the first predicate), in fact we can use this index to efficiently access the tuples that satisfy the first predicate and then verify the second condition directly on such tuples.

This process is efficient if we use the index of the most selective attribute, in fact this allows us to follow fewer pointers, thus reducing the execution time.

9.1.2 Disjunction - OR

Executing a query with a disjunction of predicates in the `WHERE` condition is more expensive, in fact we have a tuple that can verify just one of the predicates. In this case, we have two options

- If we have the indices for all the attributes in the condition then we can use them to access the tuples and verify if all the predicates are true.
- If we don't have the indices we must access the blocks sequentially.

9.1.3 Sorting

Sorting is very important and frequently used in SQL, in fact it's used not only to execute the `SORT` command but also when we want to group tuples (i.e. `GROUP BY`) or compute the union of attributes (i.e. `UNION`). When approaching sorting we can find ourselves in two situations

- The table completely fits into main memory.
- The table doesn't fit into main memory.

Table in main memory

If a table fits completely in main memory then we can simply move all the tuples in main memory and sort them using a standard sorting algorithm (like quicksort). In this case sorting is very cheap, in fact after loading all tuples the algorithm is completely executed in main memory and, as we have already said, an operation in main memory is much less expensive than an I/O operation (thus the initial I/O loading dominates the sorting).

External merge sort

If a table doesn't fit completely in main memory we have to use an algorithm that can sort the tuples only loading some of them in main memory. One of the most used and efficient algorithms is external merge sort. The algorithm is divided into multiple stages. Before describing how each stage works, consider a memory that can contain at most B blocks and that a table is made of N blocks on disk ($N > B$).

First stage In the first stage

1. Load B blocks in main memory.
2. Sort the blocks (i.e. all tuples are considered at the same time and are sorted in place). The blocks are saved into memory as a B-run (i.e. a region of memory of B blocks with elements sorted). When sorting the blocks we can use any type of sorting algorithm (e.g. quicksort, insertion sort or merge sort). Notice that a B-run is a collection of B blocks in which the tuples are sorted (e.g. `([2, 5][7, 9])`) where a tuple is represented by a number, a block is represented between square brackets and a run is represented between parentheses).
3. repeat instruction 1-3 with different blocks until all N blocks are sorted.

At the end of each state we have obtained $\lceil N/B \rceil$ B-runs.

Following stages In a stage after the first one

1. The memory is divided into B buffers (each with the dimension of a block). In particular, $B-1$ input buffers and 1 output buffer are created.
2. Each input buffer is associated with a run. The first block of each run is loaded in the respective input buffer. The output buffer is associated with a new output run (whose length is the sum of the lengths of the $B-1$ input runs). The output run is initially empty.
3. The input buffers are merged in the output buffer with the same mechanism used for merge sort. When the output buffer is full it is saved in the output run. When an input buffer is completely scanned a new block (of the same run) is loaded.
4. Instruction 3 is repeated until all the blocks of the runs have been scanned.
5. Instructions 2-4 are repeated with different input runs (different from the output runs generated by the stage).

The algorithm repeats a stage until the output run of a stage has length N. In particular stage $i+1$ is executed using the runs of the previous stage i.

Complexity To compute complexity we have to remember that main memory operations are far less expensive than input/output operations thus sorting in main memory can be neglected and we only have to count how many times a block is written to or read from disk. In particular, in every stage we have to write and read every block thus, for every stage, we have to execute $2N$ I/O operations. Now we have to understand how many stages are executed.

- The first stage is always executed.
- At every stage the number of runs is divided by $B-1$, thus the number of stages is logarithmic. The initial number of runs is $\lceil N/B \rceil$ because the first stage generates a run with B blocks.

thus the complexity of external merge sort is

$$2N \cdot (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$$

Example To better understand how this algorithm works let us consider the following example. Let us consider

- A memory with maximum capacity of 3 blocks ($B = 3$).
- A disk with 11 blocks ($N = 11$).

A block contains two tuples (a tuple is represented as an integer) and is represented between square brackets. A run is represented between parentheses. The blocks on disk are the following

```
[4, 21][3, 9][17, 19][5, 14][5, 2][7, 10][8, 6][0, 3][13, 19][21, 11][15, 1]
```

The first stage is executed as follows

1. Blocks 1 to 3 are loaded in memory and the run `([3, 4][9, 17][19, 21])` is generated.
2. Blocks 4 to 6 are loaded in memory and the run `([2, 5][5, 7][10, 14])` is generated.
3. Blocks 7 to 9 are loaded in memory and the run `([0, 3][6, 8][13, 19])` is generated.
4. Blocks 10 to 11 are loaded in memory and the run `([1, 11][15, 21])` is generated.

In the second stage we have to load two runs and generate one output runs, in particular

1. The blocks of runs 1 and 2 are loaded in memory (one at a time, replaced when a block is completely scanned) to generate run `([2, 3][4, 5][5, 7][9, 10][14, 17][19, 21])`. Let us analyse deeper how such result has been obtained.
   (a) The first block of run 1 and 2 is loaded in the input buffers. The output buffer is initialised.
   (b) Element 2 from buffer 2 is added to the output buffer.
   (c) Element 3 from buffer 1 is added to the output buffer.
   (d) The output buffer is written on disk in the output run and it is emptied.
   (e) Element 4 from buffer 1 is added to the output buffer.
   (f) A new block of the first run is loaded.
   (g) Element 5 from buffer 2 is added to the output buffer.
   (h) The output buffer is written on disk in the output run and is emptied.
   (i) A new block of the second run is loaded.
   (j) The same is repeated until both runs have been scanned.
2. The blocks of runs 3 and 4 are loaded in memory (one at a time, replaced when a block is completely scanned) to generate run `([0, 1][3, 6][8, 11][13, 15][19, 21])`.

In the third stage the runs of the second staged are merged (like in the second stage) to obtain `([0, 1][2, 3][3, 4][5, 5][6, 7][8, 9][10, 11][13, 14][15, 17][19, 19][21, 21])`

9.1.4 Join

Another critical operation is the join operation, in fact it is very expensive yet frequently used. There exists three different classes of algorithm to execute and optimise a join operation

- **Nested loops**.
- **Merge scan**.
- **Hashed join**.

Nested loops

This algorithm uses a nested loop to execute a join. Let us consider two tables `A` and `B`. To execute the join `FROM A, B ON A.Id = B.Id` we have to consider all tuples in A and for each tuple a we consider all tuples b in B. If $a.Id = b.Id$ we generate the tuple ab for the join table.

Complexity In general, this algorithm has a complexity C of

$$C = N_A + N_A \cdot N_B$$

where N_A is the number of blocks of A and B is the number of blocks of B. This cost is given by the fact that we have to access all tuples of A (thus all N_A blocks of A) and we have to access N_A times the tuples (thus the blocks) of B ($N_A \cdot N_B$).

The term N_A is dominated by $N_A \cdot N_B$ thus the cost can be written as

$$C = N_A \cdot N_B$$

If one of the two tables is small enough (say N_A is small) we can keep the smallest table in main memory and read the largest table only once (because $N_A \cdot N_B$ is done in memory). In this case we read and cache N_A blocks and we read every element of B only once, thus the cost is

$$C = N_A + N_B$$

Filter by condition When a query of a join has an additional condition (other than the condition that joins the tables) we can

1. Filter the tuples that satisfy the condition.
2. Apply nested loop on the filtered tuples.

This algorithm can be applied to different types of primary structures. In general, the algorithm works this way

1. Scan the first table.
2. If a tuple of the first table satisfies the condition, find all the tuples of the second table that have to be joined to the tuple of the first table. The tuples of the second table can be accessed using a primary structure scan or indices.

Complexity The cost of this algorithm depends on the number of tuples that have the same value (i.e. $val(Table)$). If we consider two tables A and B with a block-cardinality of N_A and N_B respectively, the cost of a join with a condition on table A is

$$C = N_A + \frac{N_A}{val(N_A)} \cdot N_B$$

As before, the first term is dominated by the second, thus the cost can be written as

$$C = \frac{N_A \cdot N_B}{val(N_A)}$$

Scan and lookup Say one of the tables involved in the join (say the second) supports an index on the attribute involved in the join (e.g. `A JOIN B ON A.att = B.att` and table `B` has an index on *att*). In this case we can

1. Do a full scan of the first table (i.e. the one not indexed, `A` in our example).
2. Use the index to access directly the tuples of `B`. Namely, if we read `ta` from a block of `A` then we can use the index on `B` to access the tuples of `B` that have `B.att = ta.att`.

In this case the join has a cost of

$$C = N_{bA} + N_{tA} \cdot C_{indexB}$$

where

- N_{bA} is the number of blocks of table `A`.
- N_{tA} is the number of tuples of `A`.
- C_{indexB} is the cost of accessing a tuple of `B` using the index.

Merge-scan join

Merge-scan can be used if tables are sorted using the same attribute key, in fact in this case we can scan the two tables in parallel and apply an algorithm like merge-sort.

Complexity This algorithm has linear complexity, in fact, both tables are scanned only once. In particular, the time complexity is

$$C = N_A + N_B$$

where N_A and N_B are the number of blocks of tables `A` and `B`.

Sorting If the primary structure isn't sorted using the same key, the engine might evaluate if sorting the table can lead to a more efficient query than other join algorithms. This happens because merge-sort is very fast (only linear complexity).

Hashed join

A join can be executed with a hashed-join only if both tables are organised in a hash structure that uses the same hash key of the attribute.

Algorithm The algorithm accesses corresponding buckets (i.e. with the same key) on the two tables.

Cost The cost of a hashed join is linear with the number of blocks of the hash tables. In particular the cost is

$$C = N_A + N_B$$

where N_A and N_B are the number of blocks of tables `A` and `B`.

9.2 Decision process

DBMS engines have to decide

- The data access operations to execute. This means that the engine has to choose if the blocks are scanned or accessed using an index.
- The order in which operations are executed (e.g., the join order).
- The implementation for an operation.
- If parallelism and pipelining can improve performance.

In particular, the reasoning process of a DBMS engine is the following

1. The engine builds a decision tree of all the possibilities and evaluates the cost of each operation (i.e. the cost of the whole query).
2. Once the best sequence of operations (i.e. a path from the root to a leaf) is decided the query is compiled.
3. The compiled query can be
   - **Saved**. In this case, if the same query is executed again, the engine can use the stored query. Periodically the stored queries are recompiled and the decision process is executed again to account for changes in the data dictionary.
   - **Deleted** after usage. In this case, if the same query is executed on the DBMS, the decision process has to be executed again.

9.2.1 SQL

SQL allows printing a visual representation of a plan built by the DBMS engine. The plan can be printed using the command `EXPLAIN PLAN` and the description of the plan contains the main steps of execution and the total time of execution so that a developer can check if a query has a bottleneck.

Part V

Triggers

Chapter 10

Introduction

In a database, tables are usually related and when some tuples of a table are added, removed or modified the related tables have to be updated accordingly. This job can be done using triggers. In particular a trigger is a set of actions executed automatically whenever an event occurs and some condition verifies. This behaviour makes databases active, in fact triggers allow a database to react and respond to events with an Event Condition Action (ECA), where

- An **event** is a `INSERT`, `DELETE` or `UPDATE` statement that modifies a table. An event fires a trigger.
- A **condition** is used to decide if the action of a trigger has to be executed.
- An **action** is a piece of code made of SQL commands and standard programming statements (like ifs, loops, ...). In other words actions have the same syntax of stored procedures.

Triggers work together with integrity constraints, in fact while triggers allow to update the database in an automatic way, constraints allow to verify that such updates are consistent.

Old and new variables When an event is executed (i.e. when the `INSERT`, `DELETE` or `UPDATE` statement modifies the table), the system keeps track of the transition between the table before and after the event. In particular such transition is represented using the variables `new` and `old` that store the modified state and the former state of the table, respectively.

Usages Triggers are usually used in databases shared between different applications to ensure central consistency. In practice we want the database to handle its consistency, otherwise the consistency checks would have had to be made on every application risking to forget some checks or implementing different rules for updating the database.

10.1 SQL

Triggers are specified in SQL using the following syntax

```
1    CREATE TRIGGER <Name>
2    { BEFORE | AFTER }
3    { INSERT | DELETE | UPDATE {OF <Column>} } ON <Table>
4    REFERENCING { [ old table [AS] <OldTableAlias> ]
```

```
5                   [ new table [AS] <NewTableAlias> ] |
6                   [ old [row] [AS] <OldTupleName> ]
7                   [ new [row] [AS] <NewTupleName> ] }
8       [ FOR EACH { ROW | STATEMENT } ]
9       [ WHEN <Condition> ]
10      <Action>
```

10.1.1 Execution mode

The `BEFORE` and `AFTER` keywords allow to specify if the event has to be executed before or after executing the action.

Before In case the `BEFORE` keyword is specified, the action is executed before the event. This pattern can be used to prevent the execution of the event if some conditions doesn't hold. In particular the sequence of events in case `BEFORE` is specified is

1. The event is evaluated but not executed, this means that the former tuples are saved in the `old` variable while the modified tuples are saved in the `new` variable. The actual table isn't modified, this means that the tuples in the table still correspond to the tuples in `old`.
2. The **condition** is evaluated.
3. The **action** is performed. The action can modify the `new` variable. This allows the trigger to correct some errors on the tuples affected by the event before modifying the actual table.
4. The **event** is executed. In particular the tuples effected by the trigger are modified using variable `new`.

Notice that the same behaviour can be obtained with the `AFTER` statement, in fact we can modify the table after the event has been executed. This operation is more expensive, in fact the table is modified twice (once by the event and once by the action) while in the former case the table was modified only by the action. Furthermore some DBMS don't allow the action to modify the table modified by the event.

After If the keyword `AFTER` is specified, the action is executed after the event, in particular the sequence of events is the following

1. The **event** is executed.
2. The **condition** is evaluated.
3. The **action** is performed.

10.1.2 Granularity of events

The keyword `FOR EACH { ROW | STATEMENT }` allows to define if the action has to be executed for every effected tuple of the table or on the whole table. In particular

- If we specify `FOR EACH ROW` then we are asking to apply the action on every tuple that is affected by the event. In this case the transition variables `old` and `new` hold a tuple (and the tuple changes for each affected row of the table) before and after applying the event on a specific row.

- If we specify `FOR EACH STATEMENT` then we are asking to apply the action on the whole table (considering only the affected rows). In this case the transition variables hold the entire table before and after executing the event.

The granularity level is set to `STATEMENT` if not specified.

Transition variables As we have seen the state of a table or of a tuple before and after an event is stored in the variables `new` and `old`. Such variables aren't available for all types of events, in particular

- If the event is a `UPDATE` event then both variables are available, in fact the database has been modified, thus a valid value existed before and exist after the event.
- If the event is an `INSERT` event then only the `new` variable is available, in fact the value added by the event wasn't present before thus we can't put it in a variable.
- If the event is a `DELETE` event then only the `old` variable is available, in fact the value deleted by the event isn't present in the current table, thus we can't put it in a variable.

These rules are valid for both row and statement levels of granularity.

10.1.3 Priority

In principle there is no limit to the number of triggers that can be written, thus it may happen that the same event fires more triggers. In such cases we have to decide the order in which the triggers are executed.

1. **before**, **statement**-level triggers are executed.
2. **before**, **row**-level triggers are executed.
3. The table is modified and **integrity constraints** are checked.
4. **after**, **row**-level triggers are executed.
5. **after**, **statement**-level triggers are executed.

It may also happen that multiple triggers with the same priority are fired. In such cases the DBMS uses an alphabetical order tie-break (if the name of trigger t_1 alphabetically comes before the name of t_2, then t_1 is executed before t_2). Another way to sort triggers with the same priority is to use the creation time. In some cases it's also possible to define some custom tie-breaking rules.

10.2 Trigger execution

When a trigger is fired, its action may modify some table `T` and fire, as consequence, a trigger associated to `T`. This can generate a cascade of triggers that may also lead to loops (e.g `T1` fires `T2`'s trigger whose action fires `T1`'s trigger). In case of loops we talk about recursive cascading.

10.2.1 Recursive cascading termination with triggering graphs

Recursive cascading is very dangerous, in fact it might lead to infinite loops. A good method to discover recursive cascading, apart from automated tools, is to build a **triggering graph**, that is a graph in which

- Each trigger is represented by a node.
- If trigger `t1` might activate trigger `t2` then node 1 is connected to node 2 with a directed arc that exits from 1 and enters in 2.

If the graph has a cycle then the execution might generate a recursive cascading and not terminated. It's important to underline that the presence of a cycle in the graph is just a sufficient condition, in fact the execution might stop even if a cycle is present (in other words we can only say that if no cycle is present than the execution surely ends). This happens because a trigger is activated only if the condition is satisfied and event when a trigger is activated, it might modify a table in a way that no trigger is fired.

10.2.2 Recursive cascading mitigation

There exist many techniques to mitigate the problem of recursive cascading and infinite loops. Oracle, for instance, defines a maximum number of action per cascade, so that if an infinite loop is present, it is stopped after a small number of iterations. MySQL doesn't allow a stored function or trigger to modify a table that is already being used (for reading or writing) by the statement that invoked the function or trigger. In other cases there is no limit to recursive cascading and it's responsibility of the programmer to avoid infinite loops.

10.3 Design principles

Triggers are very powerful but should be used with caution, in fact they can lead to unpredictable behaviours. Here is a list of things to keep in mind when we have to consider using a trigger.

- Recursive cascading triggers should be avoided unless absolutely necessary. It's also worth remembering that not all DBMS have mechanisms to stop infinite loops, thus if a trigger generates a recursive cascade, it might not stop.
- Triggers should be used only if the same thing can't be expressed using constraints. Also remember that triggers and integrity constraints can work together.
- The code executed by a trigger should be concise. If the code is very long we should consider creating a stored procedure and call the stored procedure in the trigger, instead.

10.3.1 System extensions

Since triggers have been introduced, every system has implemented some extensions to improve the expressivity of triggers. Some examples are

- **Execution modes**. Execution modes allow to specify when and how the action is executed. Some examples of execution modes are **immediate**, **deferred** and *detached* (i.e. executed in a separate transaction).

- **New type of events**. Some examples are **temporal**, **user-defined** or **system-defined** events.
- **Instead of clauses**. The instead of clause allow to replace a command with another. For instance we can execute an `UPDATE` event instead of an `INSERT` event.
- **Rule administration** to manage priority and rules.

Algorithms