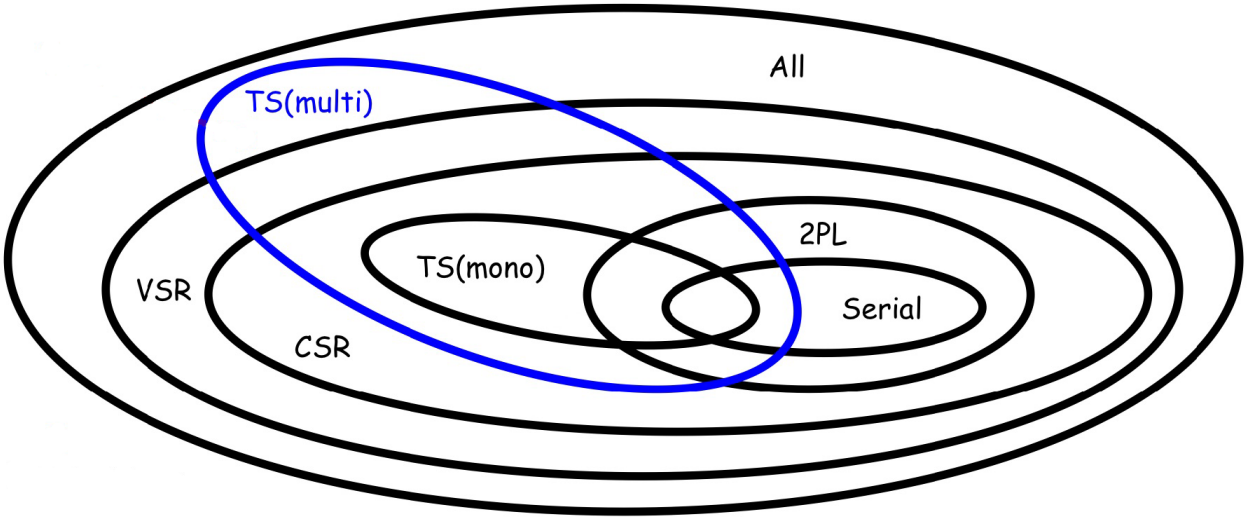
Databases 2

github.com/martinopiaggi/polimi-notes

2022-2023

Contents

1 Concurrency Control



Concurrency theory is "ideal" since it's based on the strong assumption that every transaction is submitted. Model: an abstraction of a system, object or process, which purposely disregards details to simplify the investigation of relevant properties. Operation: a read or write of a specific datum by a specific transaction.

- `r1(x)` is a read operation performed by transaction 1 on variable x.
- `w1(x)` is a write operation performed by transaction 1 on variable x.
- `r1(x)` and `r1(y)` are different operations, as they are performed by the same transaction on different variables.
- `r1(x)` and `r2(x)` are different operations, as they are performed by different transactions on the same variable.
- `w1(x)` and `w2(x)` are different operations, as they are performed by different transactions on the same variable.
- Schedule: a sequence of operations performed by concurrent transactions which **respects the order of operations** of each transaction.

A serial schedule is a schedule where each single operation of the same transaction is executed consecutively. By definition a serial schedule has not anomalies. Types of anomalies:

- Lost update: `r1 - r2 - w2 - w1` , an update is applied from a state that ignores a preceding update, which is lost.
- Dirty read: `r1 - w1 - r2 - abort1 - w2` , an uncommitted value is used to update the data.
- Non-repeatable read: `r1 - r2 - w2 - r1`, someone else updates a previously read value.
- Phantom update: `r1 - r2 - w2 - r1`, someone else updates data that contributes to a previously valid constraint.
- Phantom insert: `r1 - w2(new data) - r1`, someone else inserts data that contributes to a previously read datum.

Serial schedules are "the dream schedules" but aren't realistic, so we will talk about **serializable schedule**: a schedule where concurrent transactions are executed in a way that produces the same results as if they were executed one at a time, avoiding inconsistencies.

1.1 View-Serializability (VSR)

Two schedules are view-equivalent when they have the same operations, the same **Reads-from** and the same **Final writes**. Where:

- **Reads-from**: $ri(x)$ reads-from $wj(x)$ in a schedule S when $wj(x)$ precedes $ri(x)$ and there is no $wk(x)$ in S between $ri(x)$ and $wj(x)$.
- **Final write**: $wi(x)$ in a schedule S is a final write if it is the last write on x that occurs in S.

A schedule S is said to be view-serializable (**VSR**) if it is view-equivalent to a serial schedule of the same transactions. Bad concurrency patterns are partially rejected by view-serializability.

1.2 Conflict-equivalence (CSR)

A schedule is **conflict-serializable** if it is **conflict-equivalent** to a serial schedule of the same transactions. Two schedules are **conflict-equivalent** if:

- They contain the same operations
- All the conflicting pairs of each operation (read-write and write-write conflicts) of the transactions occur in the same order.

The class of conflict-serializable schedules is named CSR. Since a **schedule is in CSR iff its conflict graph is acyclic**, we use a conflict graph that has:

- One node for each transaction T_i
- One arc from node T_i to T_j for each conflict between an operation o_i of T_i and an operation o_j of T_j such that o_i precedes o_j (note that a conflict is between two operations where at least one of them is a **write** operation... no conflict between two reads obviously).

1.3 Timestamp Concurrency Control

In practice we can´t evaluate the schedules online/real-time and say if they are CSR or VSR. So we can use locks (which it's called "pessimistic tecnique" since it eventually makes the requester waiting) or optimistic tecniques based on timestamps and versions.

1.3.1 Ts Mono

The scheduler receives read/write requests tagged with the timestamp of the requesting transaction . You use the age in order to resolve potential conflicts. We reject if:

- $R_{ts}(x)$ we reject if $ts < WTM(x)$
- $W_{ts}(x)$, we reject if $ts < RTM(x)$ or $ts < WTM(x)$

Where WTM and RTM are respectively the last write and the last read (indicated with timestamps). There is also a variant called "Thomas Rule" to basically reduce the killed transactions. The only difference of Thomas Rule is to no kill the write operations if $ts < WTM(x)$ with the motivation that our write is "obsolete" and it can be simply skipped without killing the entire transaction. That's a fair consideration: we are skipping a write on an object that has already been written by a younger transaction. Note that this versions are not really implemented, they are just a little step to build some fundamentals for TS Multi.

1.3.2 TS Multi

TS Multi is basically TS Mono multi-versioned. Since every DBMS have to guarantee an 'undo command' it's very 'natural and logic' use a multi-versioned system (It's used actually).

> Idea: each write generate new versions, reads access the "right" version. $WTM_1(x) \dots WTM_N(x)$ are the versions, sorted from oldest to youngest.

$r_{ts}(x)$ is always accepted. A copy x_k is selected for reading such that: - If $ts \geq WTM_N(x)$, then $k = N$ - else $WTM_k(x) \leq ts < WTM_{k+1}(x)$

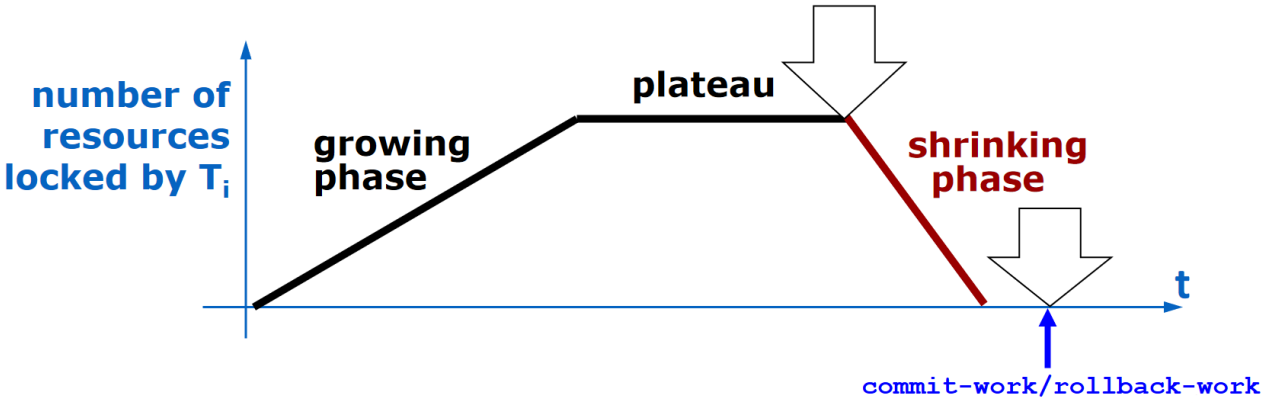
$w_{ts}(x)$: - If $ts < RTM(x)$ the request is rejected - Else a new version is created for timestamp ts and N is incremented. NB!! For simplicity (in the exercises) we will kill also if $ts < WTM(x)$

We are complicating the implementation but in change we now have a huge benefit: **the read will be always accepted**.

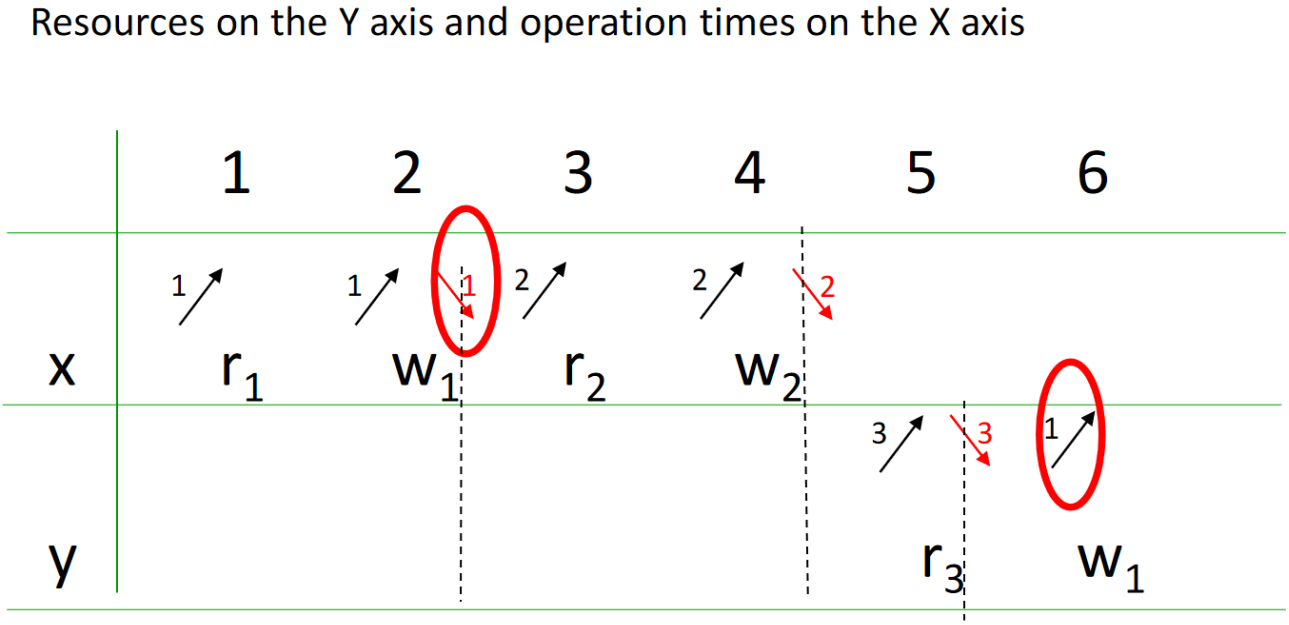
1.4 2 Phase Locking (2PL)

The rules of 2PL as a system of constraints imposes precedence rules upon the lock and unlock requests: basically each lock on any resource of the same transaction has to follow the "2 phase lock" rule. Note that:

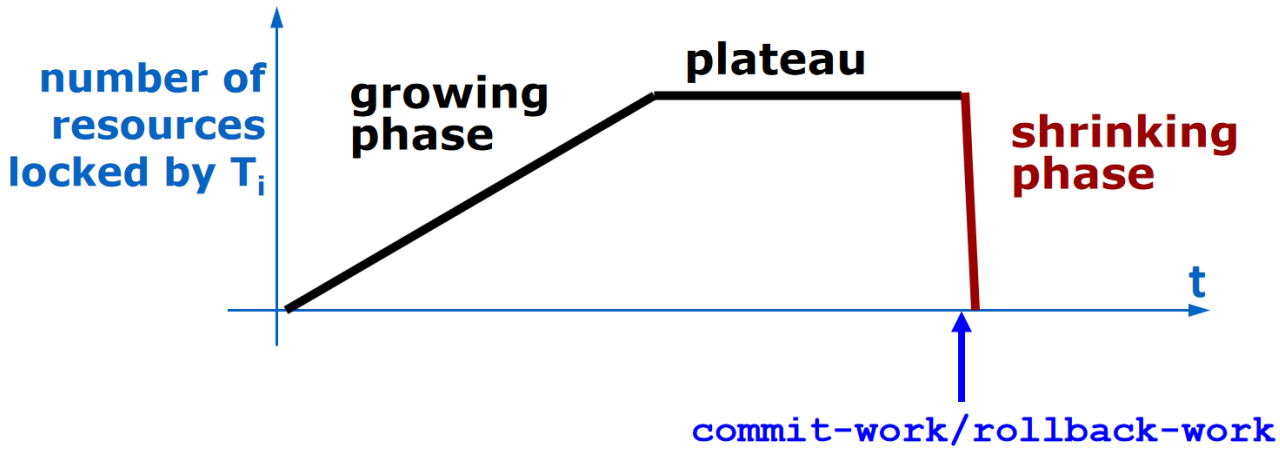
- In the same resource **only shared locks can coexist**.
- In the same transaction all the **locks must precede the unlocks**: showing an evident growing phase, a plateau and a shrinking phase.



Last phrase written again: 2PL imposes to **not** acquire any lock after the first release: in this way we would have a "growing phase" where all the locks are acquired, a plateau and then all the releases in a shrinking phase and the final commit/rollback.



1.4.1 Strict 2PL



The strict 2PL has the same "two-phase rule" with a little variation: locks held by a transaction can be released only after commit/rollback.

Extra: actually the Strict 2PL prevents all possible anomalies except "Phantom Insert" anomaly. To prevent this inconsistency, a technique called "predicate locks" can be used, which extends the notion of data locks on future data using a predicate that blocks all data that satisfies it.

As you know after 3 years of Computer Science when there are locks there are deadlocks and starvation. There are mainly three possible way to solve this:

- Timeout: killing transactions after its timeout runs out
- Deadlock prevention: killing transactions which could lead to deadlock
- Deadlock detection: kill one of the transactions in a deadlock that occurred.

The two main rules to reduce the deadlocks frequency are:

- use Update Locks
- use Hierarchical Locks

We we will see a super simple algorithm to detect cycles in wait-for-graph, in particular in a "distributed systems" environment: Obermark's algorithm.

1.4.2 Update Locks

Same locks but with the introduction of Update Locks (UL):

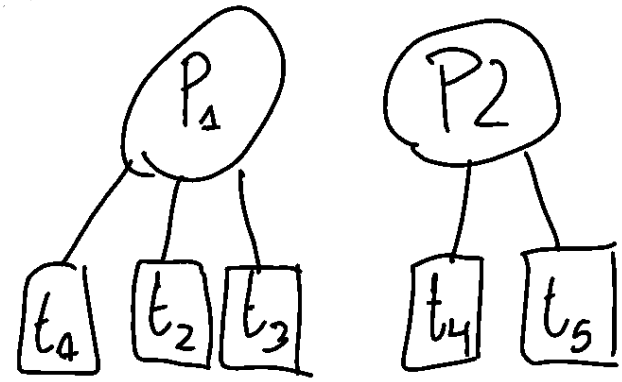
- UL is used only if a write is needed later.
- Obviously the scheduler uses UL and it can't get a SL, release it and then get XL later!

Clearly deadlocks are possible in the presence of UL but less frequently. Indeed, UL only makes deadlock less likely, by preventing one type of (very frequent) deadlock, due to update patterns, when two transactions compete for the same resource. If we consider two distinct resources X Y, and two transactions that want to access them in this order:

$$r1(X)r2(Y)w1(Y)w2(X)$$

It is likely that they end up in deadlock, especially if the system on which they run applies 2PL. UL is totally irrelevant here, because there is no update pattern.

1.4.3 Hierarchical Locks



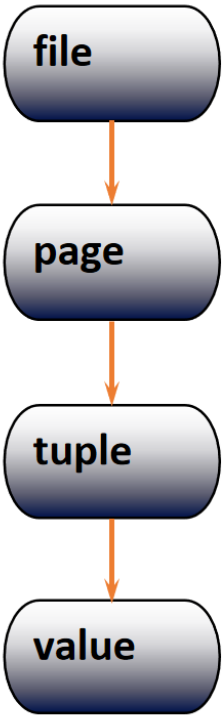
Generally all the exercises on Hierarchical Locks are in a 2PL or 2PL-strict environment. Basically are locks (actually are locks really used in reality and in practice applications) where there are **more levels** with difference importance (like a hierarchy). In a data structure, if you want to access an item, you will perform an "intentional lock" over all the containers of that item. In particular the scheduler can use:

- ISL: Intention of locking a subelement of the current element in shared mode.
- IXL: the same of ISL but in exclusive mode
- SIXL: Lock of the element in shared mode with intention of locking a subelement in exclusive mode (SL+IXL)

General rules:

- Intentional shared/exclusive lock over the container
- shared/exclusive lock over the actual target

As always shared is compatible with others shared and with an intentional shared lock. **Different intentional locks on the same resource are always possible**.



1.4.4 Obermark's algorithm

The algorithm is a distributed one and it communicates the waiting sequences to other instances of the same algorithm. Consider any waiting condition where a sub- transaction t_i , activated by a remote DBMS on node h, is waiting for a lock to be released by another transaction t_j which in turn waits for the termination of a remote sub-transaction invoked on a DBMS on node k. The waiting situation is summarized by the following wait sequence:

$$EXT_h \rightarrow t_i \rightarrow t_j \rightarrow EXT_k$$

Messages are sent only " 'ahead"'. That is, towards the DBMS on node k, where the sub-transaction for which t_1 is waiting was activated. Also, the message is sent only if $i > j$. Also every message, sent from an instance to another one, is"compressed":

$$E_h \rightarrow t_i \rightarrow t_m \rightarrow t_n \rightarrow ... \rightarrow t_p \rightarrow t_j \rightarrow E_k$$

It's compressed into the message

$$E_h \rightarrow t_i \rightarrow t_j \rightarrow E_k$$

and is sent to node k iff $i > j$. Note that this is just a convention, the "main algorithm" just propagates local waiting condition to remote nodes and it could work also with the "opposite conventions": send only if $i < j$.

2 SQL Triggers

Main paradigm to reason:

```
EVENT -> CONDITION -> ACTION
```

- Use triggers to guarantee that when a specific operation is performed, related actions are performed
- Do not define triggers that duplicate features already built into the DBMS. For example, do not define triggers to reject bad data if you can do the same checking through declarative integrity constraints
- It is essential to enforce database constraints at the database level as opposed to at the application level. This is because multiple applications may access and manipulate the same database. By implementing constraints at the database level, it ensures that data remains consistent and organized across all applications that access the database, thereby preventing inconsistencies and disorganization.
- Limit the size of triggers to 60 lines of code or less
- Use stored procedures to contain most of the code and call them from the trigger
- Use triggers only for centralized, global operations that should be fired for the triggering statement, regardless of the user or application that issues the statement
- Avoid recursive triggers if not absolutely necessary, as they may cause the DBMS to run out of memory
- Use triggers to guarantee properties of data that cannot be specified by means of integrity constraints

2.1 Termination analysis

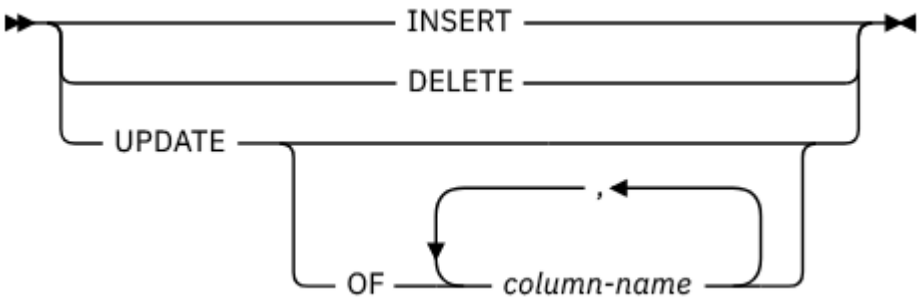
Triggers are powerful but also complex due to cascading policies which can lead to bugs and recursive cascading. Some systems block cascading of triggers to prevent these issues. Cascading is generally not good and in some DBMS is even forbidden. But it's important to remember that triggers are powerful and can be useful if used correctly. You can do termination analysis using a triggering graph:

- Each node is a trigger
- Each arc from trigger i to trigger j is present only if T_i effects may activate T_j

2.2 Syntax

Main structure to make triggers:

```
create trigger <trigger_name>
{before | after}
{insert | delete | update [of <column>]} on <table>
[referencing (to put aliases on old, new, etc)]
[for each {row | statement}]
[when condition]
BEGIN
[my action]
END
```



2.2.1 Some useful SQL functions when using triggers

- `COALESCE` function returns the first non-NULL value in a list of expressions. It can be used to provide a default value when a `NULL` value is encountered.

```
SELECT COALESCE(NULL, 'Tony Montana', 'Scarface') AS "Main Character";`
```

- `IS NULL/IS NOT NULL` operators are used to test for `NULL` values in a column or expression

```
SELECT * FROM movies WHERE main_character IS NULL;
```

- `EXISTS` operator is used to check if a row exists in a subquery.

```
SELECT * FROM movies
WHERE EXISTS
(SELECT 1 FROM box_office WHERE box_office.movie_id = movies.id AND box_office.revenue > 10000);
```

- `IFNULL` and `DECLARE` common pattern:

```
DECLARE X AS INTEGER
IFNULL(<SELECT_QUERY_HERE>, 0) INTO X

IF (X<>0)
THEN
    \\
ELSE
    \\
ENDIF;
```

- It is possible to add values for all the columns of a table using the `INSERT INTO` statement.

```
INSERT INTO _table_name_
VALUES (_value1_, _value2_, _value3_, ...);
```

- A `REFERENCING` clause in a SQL trigger is used to specify the name of a transition table that is used to hold the old and/or new values of the rows that are affected by the trigger.
- `FOR EACH ROW` just says to execute the trigger body for each of the matched and affected table rows.

3 JPA

ORM stands for Object-Relational Mapping, and it's basically a way to interact with a relational database using an object-oriented programming model. Instead of writing SQL queries to get data from a database, you can just use objects in your code.

One of the most popular ORM frameworks for Java is JPA (Java Persistence API). JPA is basically the standard for ORM in the Java Enterprise Edition (Java EE) platform, and it makes working with a relational database really simple. You can do all the basic stuff like creating, reading, updating, and deleting data without having to write complex SQL queries. Plus, it has some awesome features like caching and lazy loading that can help you make your code run faster.

JPA Introduction - javatpoint

JPA auto-generates SQL code from Java code, creating correspondence between Java objects and DB tuples. This is called ORM (Object-relational mapping):

- classes are tables
- objects instances are tables rows (tuples)
- the physical memory address of an objects is the primary key of the tuple

- references to other objects are the foreign keys
- methods are triggers

By default entities are mapped to tables with the same name and their fields to columns with the same names, but you can always change it using some annotations:

```
@Entity @Table(name="SUPERHEROS")
public class Superhero {
@Column(name="REAL_NAME",nullable=false)
private String real_name;
...
}
```

3.0.1 Relationships

All relationships in JPA are unidirectional, the only way to make bidirectional relationships is using two 'paired' unidirectional mappings. The types of relationships are:

- Many-to-one
- One-to-many
- One-to-one
- Many-to-many

3.0.2 Fetch Policy

By default, single-valued relationships are fetched eagerly and collection-valued relationships are loaded lazily. However, you can specify the fetch mode using the `fetch` attribute in annotations such as `@ManyToOne` and `@OneToMany`.

- `@ManyToOne(fetch = FetchType.EAGER)` is used when there aren't many elements and you can load them runtime when needed.
- `@OneToMany(fetch = FetchType.LAZY)` viceversa of `EAGER`.

3.0.3 Cascade Types

Cascade types define the behavior of entities when other entities are persisted, removed, merged, etc. The available cascade types are:

- PERSIST
- REFRESH
- REMOVE
- MERGE
- DETACH

3.1 JPA relationships

In JPA, relationships are implemented by a foreign key column that refers to the key of the referenced table. This column is called a join column. The annotation `@JoinColumn` indicates the foreign key column that implements the relationship in the database, and is normally inserted in the entity owner of the relationship, which is the one mapped to the table that contains the foreign key column.

When specifying bidirectional relationships, the `mappedBy` attribute indicates that "this side" of the relationship is the inverse of the relationship, and the owner resides in the "other" related entity. In the absence of the `mappedBy` parameter, JPA creates a bridge table, which is used for many-to-many relationships. The purpose of the `mappedBy` parameter is to instruct JPA not to create a bridge table, as the relationship is already being mapped by a foreign key in the opposite entity of the relationship.

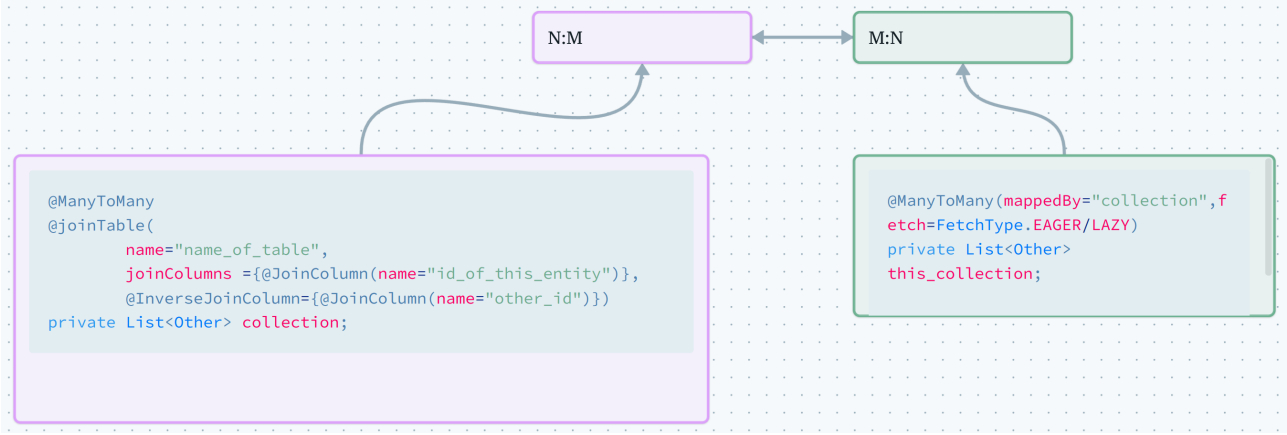
3.1.1 One to many

```
@Entity
public class B{
@Id private int id
@OneToMany(mappedBy = "java_var")
private Collection<A> As;
...
}

//On the other side:

@Entity  //this is the owner of the relationship
public class A{
@Id @GeneratedValue(strategy=GenerationType.AUTO)
private int id;

@ManyToOne
@JoinColumn (name="fk") //the column where there is the foreign key
private B java_var;
...
}
```

3.1.2 One to One

```
@OneToOne(mappedBy="B");
private Class A;

// owner side:
@OneToOne(mappedBy="A");
private Class B;
```

3.1.3 Many to many



The logical model of a N:M relationships requires a join table (aka bridge table). The N:M relationship with attributes between a parent product and its sub-products is mapped differently in the two directions.

```
@Entity
public class B{
```

```
@Id private int id
@ManyToMany(mappedBy = "java_list_of_Bs")
private List<A> As; //actually not only lists .. but any kind of collections
...
}

//On the other side:

@Entity
public class A{
@Id @GeneratedValue(strategy=GenerationType.AUTO)
private int id;

@ManyToMany
@joinTable(
    name="table_name",
    JoinColumns = @JoinColumn(name="this_entity_fk"),
    @InverseJoinColumn = @JoinColumn(name="other_fk"))
private Collection<B> java_list_of_Bs;
...
}
```

Note that the owner in these cases is not relevant.

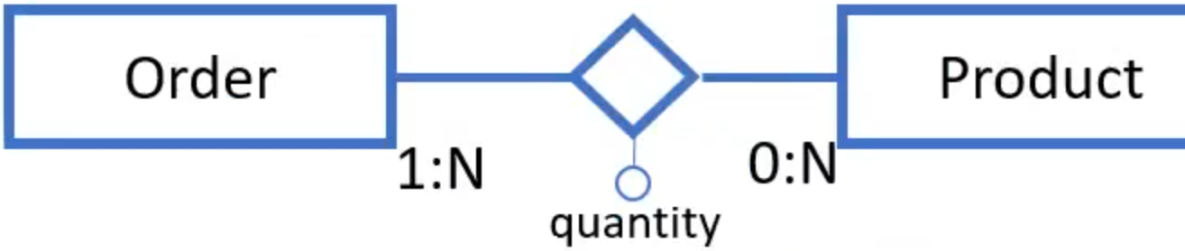
3.1.4 ManyToMany with attributes

- In a "regular" N:M relationship, the access paths are:
  - Fetch all instances of A given B
  - Fetch all instances of B given A
- In N:M with attributes, there are additional access paths:
  - The values of the attributes of the pair `<A, B>`

In every collections implemented as tuples (e.g., `<Product, quantity>`)) there are some differences with the "standard" `@ManyToMany` relationship:

- Entities key element collection are specified with the `@ElementCollection` annotation.
- We also use a `@CollectionTable` annotation
- We map the join from father to child using the `@JoinColumn` to specify the column holding the father id (like in `@ManyToMany`)
- Instead of specifying the `@inverseJoinColumns`, we use a `@MapKeyJoinColumn` annotation to specify the column holding the PK of the entity used as a map index

```
@ElementCollection(fetch=FetchType.TYPE)
@CollectionTable(name="id_name",
                 joinColumns =
                 @JoinColumn(name="this_id")
)
@MapKeyJoinColumn(name="other_id")
@Column(name="third_attr_in_relation")
private Map<Other,Attribute> map;
```



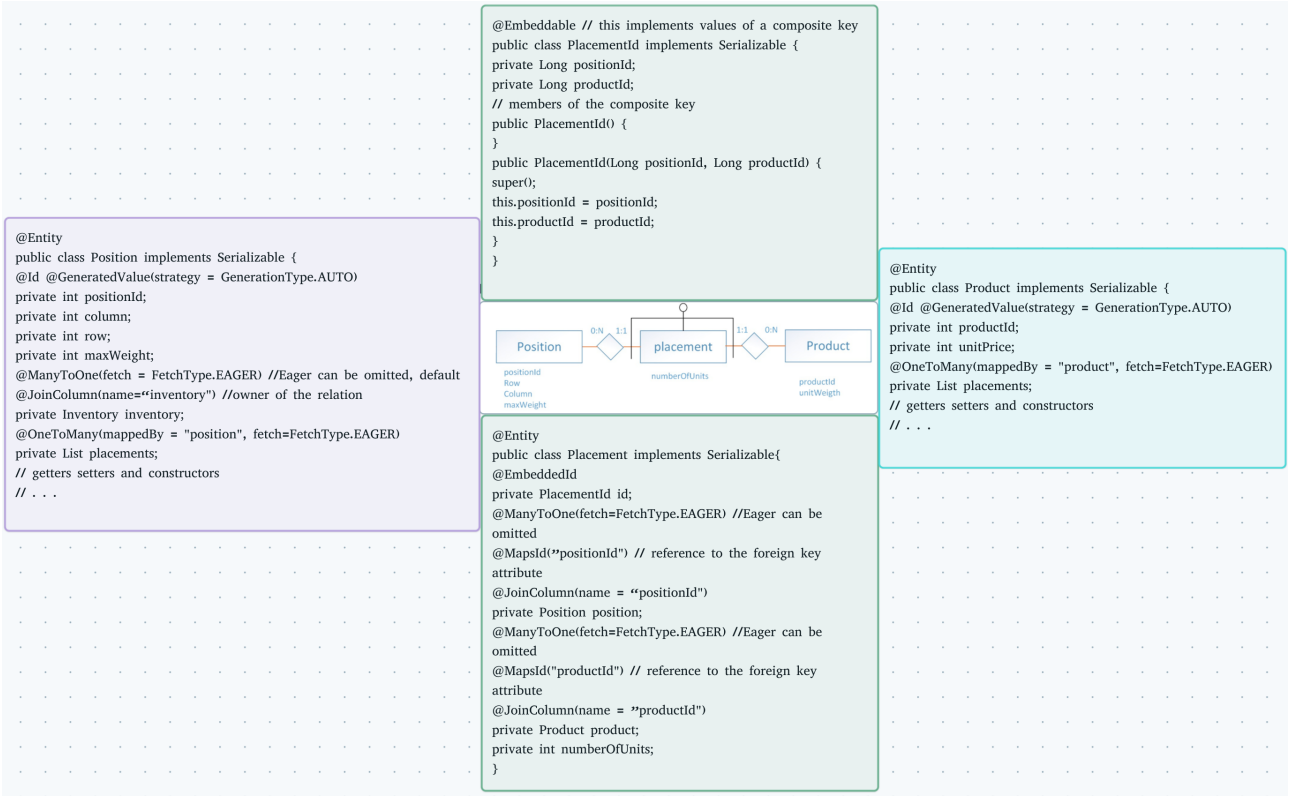
3.1.4.1 Example

We can't have a traditional join table with 2 columns because we have 3 attributes: so we use a map `Map<Product, Integer>` .

3.1.5 Composite Keys

A relation with a composite key is implemented by creating an `@Embeddable` class that represents the composite keys. This class should have the attributes of the ids that make up the composite key, and a constructor that takes these attributes as parameters and initializes them.

Then, this class is linked to the interested entity using the `@EmbeddedId` annotation.



So the recap is: you have to use the keyword `@Embeddable` (over the class) and a `@EmbeddedId` over the primary key of this new bridge entity, which will also contain the ID of the other two entities using `@MapsId("id1")` before `@JoinColumn(name = "id1")` and `@MapsId("id2") ...` .

3.1.6 Extra: OrderBy

You can specify an ordering every time you find in the exercise something like "The application permits the operator to select .... ordered by submission **date descending**" using `@OrderBy` :

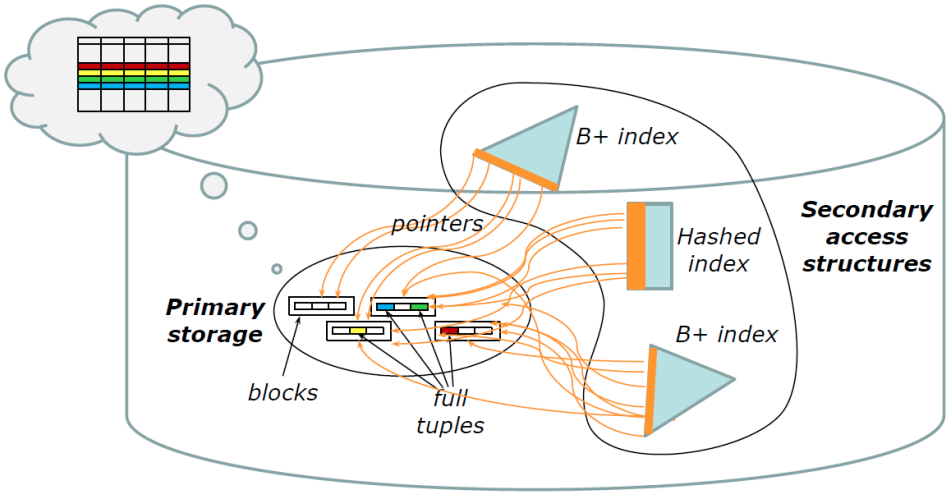
```
@OrderBy("submissionDate DESC")`
```

4 Physical Databases and Query Optimizations

We can divide structures in two classes:

- Primary structure is the main storage mechanism of a table and contains all the tuples of the table. Its main purpose is to store the table content.
- Secondary structures are used to index primary structures and only contain the values of certain fields.

Files are made up of "physical" components called blocks, while tables are made up of "logical" components called tuples. The size of a block is typically fixed and depends on the file system and disk formatting, while the size of a tuple (also known as a record) is variable and depends on the database design.



Operations are performed in main memory and affect pages (in the buffer). These operations include:

- Insertion and update of a tuple
- Deletion of a tuple
- Access to a field of a particular tuple, which is identified according to an offset w.r.t. the beginning of the tuple and the length of the field itself (stored in the page dictionary).

4.1 Data structures

Three main types of data access structures:

- Sequential structures
- Hash-based structures
- Tree-based structures

	Primary	Secondary
Sequential structures	Typical	Not used
Hash-based structures	In some DBMSs (e.g., Oracle hash clusters, IDM DB2 "organize by hash" tables)	Frequent
Tree-based structures	Obsolete/rare	Typical

4.1.1 Sequential structures

The entry-sequenced sequential structure is effective for:

- Insertion, which does not require shifting (The default setting for a data structure is not to sort data, and as such, a shifting operation will never happen.)
- Space occupancy, as it uses all the blocks available for files and all the space within the blocks
- Sequential reading and writing (e.g. `SELECT * FROM T`), especially if the disk blocks are contiguous (reducing seek and latency times)
- Only if all (or most of) the file is to be accessed

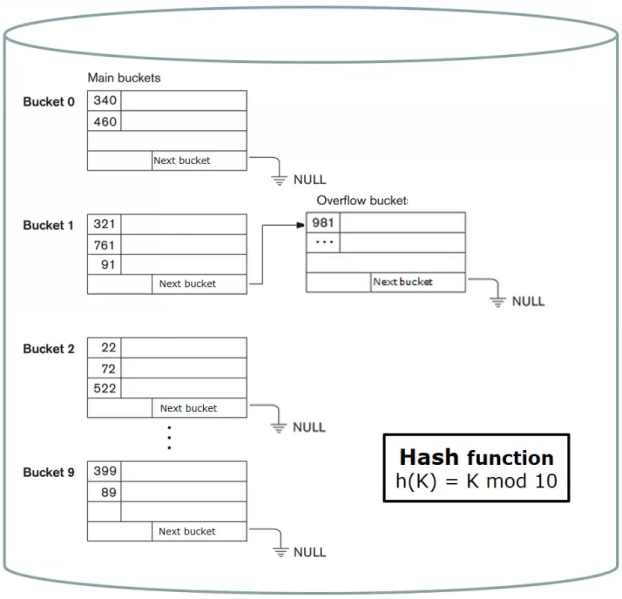
However, this structure is non-optimal for:

- Searching specific data units (e.g. `SELECT * FROM T WHERE ...`) as it may require scanning the whole structure. However, with indexes, it can be used more efficiently.
- Updates that increase the size of a tuple, as "shifts" are required. This shift may also require storage in another block.

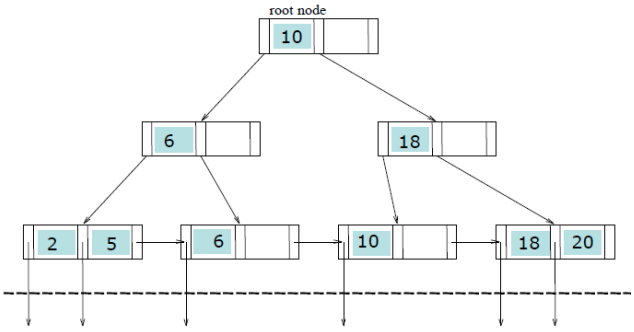
	Entry-sequenced	Sequentially-ordered
INSERT	Efficient	Not efficient
UPDATE	Efficient (if data size increases $\rightarrow$ delete + insert the new version)	Not efficient if data size increases
DELETE	"Invalid"	"Invalid"
TUPLE SIZE	Fixed or variable	Fixed or variable
SELECT * FROM T WHERE key ...	Not efficient	

4.1.2 Hash based structures

Good for point query and bad for range queries (because the lack of sorting).



4.1.3 B+



The B+ tree is considered the most commonly used method for indexing data. As a multi-level index with one root and multiple intermediate and leaf nodes, each node is stored in a block and boasts a large fan-out (a high number of children).

4.1.4 Cost of queries and query optimization

Queries are evaluated based on their costs, which is determined by the number of IOPs required to execute them. We can estimate the cost of: - equality, conjunction and disjunction predicates - sorting and interval queries reasoning about the data structures properties and also the statistic on data contained in each structure. Join operations can be done using different strategies, with costs that depend on the properties of the tables being joined:

- Nested loops: a "while cycle" on the external table for each block on the internal table. The cost is $ext_{blocks} + ext_{blocks} * int_{blocks}$. The smallest table is always the used as the external one and if a table is small enough, it is cached (loaded in (main) memory) so that the total cost is $ext_{blocks} + int_{blocks}$.
- Scan and lookup: if one table supports indexed access in the form of a lookup: the cost is $ext_{blocks} + ext_{tuples} * int_{lookup/spacecost}$.
- Merge-scan join: possible if **both** tables are ordered by the attribute used in the join predicate. The cost is linear in blocks of the two tables.

- Hashed join: possible if both tables are hashed by the attribute used in the join predicate. The cost is linear in blocks of the two tables.

5 Ranking

Ranking is a method to return approximate matches to a query, based on relevance. It's a shift from exact queries to approximate ones where the result is not necessary an exact match. Results are returned in an order of relevance determined by ranking criteria or a scoring function. This "score" can be determined by a deterministic function or by a parametric function of some object attribute values weighted by parameters that can be subjective or even unknown. Because of this, ranking is not sorting. Sorting is deterministic and not based on preferences, while ranking is based on preferences. Note that the boundary between them is not absolute. The actual subject of the exam are:

- Rank aggregation: This method combines several ranked lists of objects in a robust way to produce a single consensus ranking.
- top-k queries: This method extracts the top k objects according to a given quality criterion (typically a scoring function).
- Skyline queries: This method extracts the subset of objects that are not dominated by other objects with respect to multiple quality criteria.

5.1 Ranking aggregation

Voter 1	Voter 2	Voter 3	Voter 4	Voter 5
Rank1	Rank2	Rank3	Rank4	Rank5
A	B	D	E	C
B	D	B	A	E
C	E	E	C	A
D	A	C	D	B
E	C	A	B	D

Borda's proposal, also called the Borda count, assigns points to objects based on their ranking. The object with the highest overall points wins. Condorcet's proposal, also known as the Condorcet method, pits objects against each other in head-to-head "matches" and the winner is determined by the object who wins the most matches.

Both Borda and Condorcet methods aim to produce fairer results by considering all voters' preferences but they have limitations.

5.1.1 MedRank

Other methods for rank aggregation only use the position of objects in the list, without any additional scores or weights. One such method is MedRank, which is based on the concept of a median and is an effective technique for combining opaque rankings.

Input: integer k, ranked lists $R_1, \ldots, R_m$ of n elements Output: the top k elements according to median ranking 1. Use sorted accesses in each list, one element at a time, until there are k elements that occur in more than

$m/2$ lists 2. These are the top k elements

An optimal algorithm has a cost of execution that is never worse than any other algorithm on any input. MedRank is not optimal, but it is instance-optimal, meaning that it is the best possible algorithm up to a constant factor on every input instance that accesses the lists in sorted order which means that its cost cannot be arbitrarily worse than any other algorithm on any problem instance.

5.2 Top-k

Top-k, aka ranking queries, aim to retrieve the top-k best objects from a large set, where best means most relevant. This can be useful in various applications, such as Information Retrieval, Recommender Systems, and Data Mining.

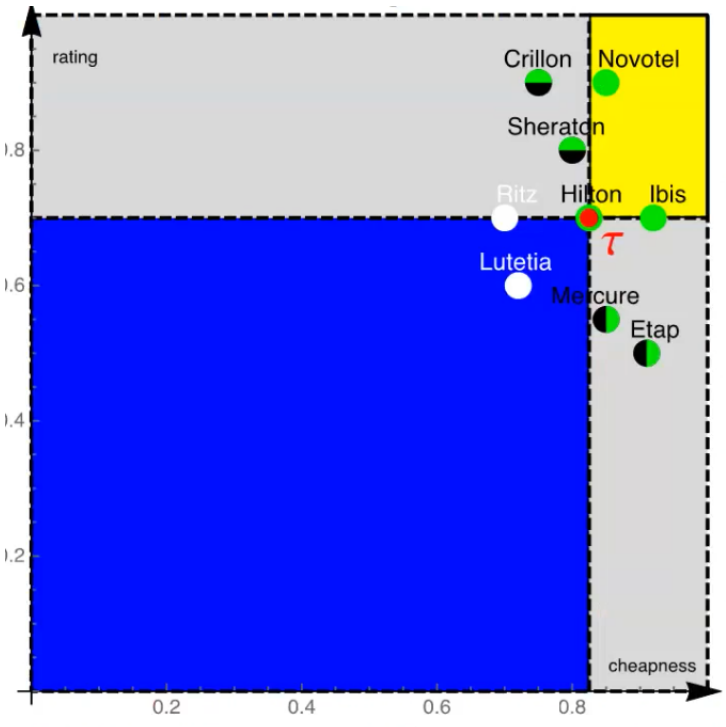
5.2.1 B0 algorithm

Input: integer $k > 1$, ranked lists $R_1, ..., R_n$ Output: the top-k objects according to the MAX scoring function

1. Make exactly k sorted accesses on each list and store objects and partial scores in a buffer
2. For each object in the buffer, compute the MAX of its (available) partial scores
3. Return the k objects with the highest scores.

5.2.2 FAGIN's algorithm

1. Make k sorted accesses in each list until there are at least k objects in common
2. For each extracted object, compute its overall score by making random accesses wherever needed
3. Take the k objects with the best overall score from the buffer



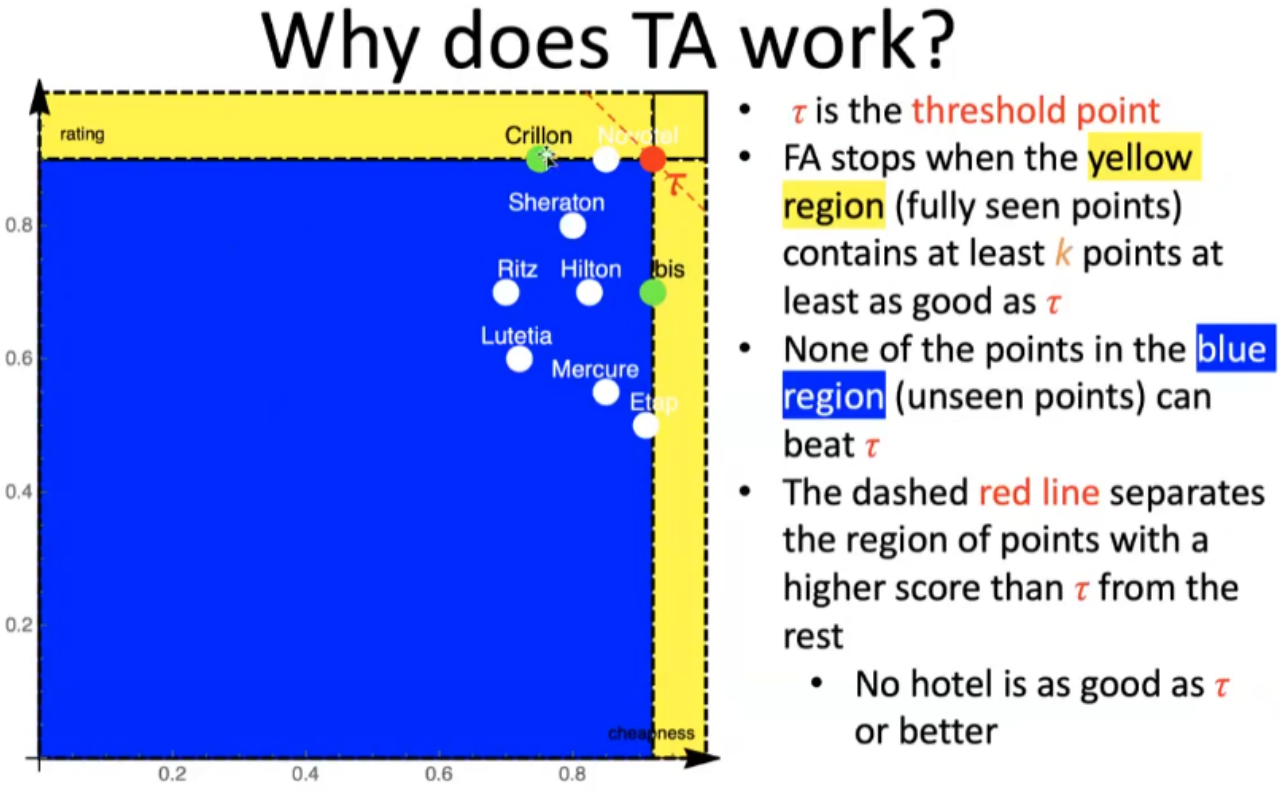
- The threshold point (τ) is the point with the smallest seen values on all lists in the sorted access phase
- FA stops when the yellow region (fully seen points) contains at least k points
- The gray regions contain the points seen in at least one ranking
- None of the points in the blue region (unseen points) can beat any point in the yellow region

5.2.3 Threshold algorithm

1. Do a sorted access in parallel in each list $R_1 \ldots R_n$

2. Don´t wait to do random access like FA, but do immediately and extract the score S
3. Define threshold $T = S(s1, ..., sn)$, where s_i is the last value seen under sorted access for list i
4. If the score of the k_{th} object is worse than T, go to step 1 otherwise return the current top-k objects

TA is an instance-optimal algorithm, which means that it is guaranteed to find the best possible results among all algorithms that use the same kind of accesses, but it's not guaranteed to be optimal among all algorithms in general. The authors of this algorithm received the Gédel prize in 2014 for the design of innovative algorithms.



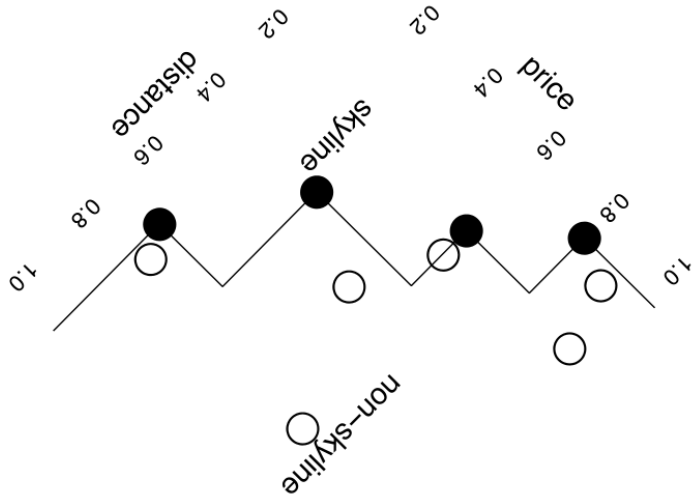
5.2.4 NRA algorithm

NRA uses only sorted accesses over the lists.

1. Make a sorted access to each list
2. Store in B each retrieved object o and maintain $S(o)$ and $S^*(o)$ and a threshold r
3. Repeat from step 1 as long as $S^-(B[k]) < \max\{\max\{S^+(B[i]), i > k\}, S(\tau)\}$

Algorithm	scoring function	Data access	Notes
B_(0)	MAX	sorted	instance-optimal
FA	monotone	sorted and random	cost independent of scoring function
TA	monotone	sorted and random	instance-optimal
NRA	monotone	sorted	instance-optimal, no exact scores

5.3 Skyline



The skyline is the collection of objects that are ranked as the best according to a possible monotone scoring function. Skyline is based on the idea of dominance. Tuple t dominates tuple s if t is not worse than s in any attribute value and is better than s in at least one attribute value. Typically, lower values are considered better.

	Ranking queries	Skyline queries
Simplicity	No	Yes
Overall view of interesting results	No	Yes
Control of cardinality	Yes	No
Trade-off among attributes	Yes	No

A point is considered to be in the n-skyband if it is not dominated by more than n tuples. The skyline is a 1-skyband because all of its objects are not dominated. **To be part of the skyline, a tuple must be the top-1 result according to at least one monotone scoring function.**

5.3.1 BNL - Block Nested Loop

Input: a dataset D of multi-dimensional points Output: the skyline of D

```
let W = []
for each point p in D:
    if there is no point q in W that dominates p:
        remove from W all points that p dominates
        add p to W

return W
```

W=Window

The time complexity of the algorithm is still $O(n^2)$.

5.3.2 SFS - Sort Filter Skyline

Input: a dataset D of multi-dimensional points Output: the skyline of D

```
Let S = D sorted by a monotone function of D's attributes
Let W = []
for every point p in S
    if p not dominated by any point in W
        add p to W
return W
```

If the input is sorted, then a later tuple cannot dominate any previous tuple: pre-sorting pays off for large datasets. The time complexity of the algorithm is still $O(n^2)$.