Indexing and Query optimization

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Lecture topics

- Query optimization.
- Examples of slow query operations.
- Hashing.
- Trees.

Query optimization

Reasons

- Query needs to be fast.
- Sometimes they are not.
- You do not want to see your nephew born before retrieving the book you are looking for from Amazon.

Query optimization

Causes

- Too much data (Big data analysis).
  - Data clustering.
  - Better hardware. (arrays of disks, caching, ...)
- Too complex queries (DBMS optimization)
  - Refactor query. (Access planner)
  - Refactor data. (Indexing)

Query optimization

Indexes

- Query refactoring not always possible.
- Build additional data to speed up the data retrieval.

Query optimization

Indexes

- Take your text book and look for the paragraph titled "Key constraints" without using the index. How many pages have you looked?

Query optimization

Indexes

- Take your text book and look for the paragraph titled "Key constraints" without using the index. How many pages have you looked?
- **Answer:** 29 (from page 3 to 32).

Indexes

- Take your text book and look for the paragraph titled "Key constraints" without using the index. How many pages have you looked?
- **Answer:** 29 (from page 3 to 32).
- Do the same using the index. How many pages have you looked?

Query optimization

Indexes

- Take your text book and look for the paragraph titled "Key constraints" without using the index. How many pages have you looked?
- **Answer:** 29 (from page 3 to 32).
- Do the same using the index. How many pages have you looked? **Answer:** 2 (1 in the index, 1 in the text).

Where


```sql
SELECT name
FROM ships
WHERE firepower = 1500
```

ships				
name	type	firepower	speed	position
Red 1	X-Wing	10	300	(1,3,1)
Red 2	X-Wing	10	300	(1,2,1)
Red 3	X-Wing	10	300	(0,2.5,1)
Red 4	X-Wing	10	300	(2,2.5,1)
Red 5	X-Wing	10	300	(2,2.5,0)
Red 6	X-Wing	10	300	(1,2.5,0)
Tantine IV	Corellian Corvette	60	300	(4,2.5,0)
Tyranny	Imperial Star Destroyer	1500	100	(12,0,0)
Accuser	Imperial Star Destroyer	1500	100	(-12,0,0)
Bombard	Victory Star Destroyer	1500	175	(-6,1,0)

Where


```sql
SELECT name
FROM ships
WHERE firepower = 1500
```

ships				
name	type	firepower	speed	position
Red 1	X-Wing	10	300	(1,3,1)
Red 2	X-Wing	10	300	(1,2,1)
Red 3	X-Wing	10	300	(0,2.5,1)
Red 4	X-Wing	10	300	(2,2.5,1)
Red 5	X-Wing	10	300	(2,2.5,0)
Red 6	X-Wing	10	300	(1,2.5,0)
Tantine IV	Corellian Corvette	60	300	(4,2.5,0)
Tyranny	Imperial Star Destroyer	1500	100	(12,0,0)
Accuser	Imperial Star Destroyer	1500	100	(-12,0,0)
Bombard	Victory Star Destroyer	1500	175	(-6,1,0)

Number of comparisons: 10

WHERE

Indexes

- How many comparisons we do at most in a table with R records?

WHERE

Indexes

- How many comparisons we do at most in a table with R records?
- **R comparisons**

WHERE

Indexes

- How many comparisons we do at most in a table with R records?
- **R comparisons**
- How many comparisons we do at least in a table with R records?

WHERE

Indexes

- How many comparisons we do at most in a table with R records?
- **R comparisons**
- How many comparisons we do at least in a table with R records?
- **R comparisons**

WHERE

Indexes

- How many comparisons we do at most in a table with R records?
- **R comparisons**
- How many comparisons we do at least in a table with R records?
- **R comparisons**
- Selection always requires to scan the entire table.

SORTING

- Sorting and grouping requires to sort the column values.
- The best sorting algorithm requires about $R \log R$ operations, where R is the number of records.
- Running the query below requires about $10 * \log 10 \simeq 23$ operations.

```sql
SELECT type
FROM ships
WHERE firepower = 500
ORDER BY name DESC
```

JOIN

- Generate pairs with one element from the first table and the second from the other followed by a selection.
- Same problem of the selection.
- Consider the following query applied to ship and the table below.

```
SELECT s.name,p.damage
FROM ships s,projectiles p
WHERE s.position = p.position AND
      s.name = p.target
```

Projectiles		
target	position	damage
Red 3	(0,1,0)	30
Red 3	(3,1,-2)	50
Red 3	(0,2.5,1)	100

JOIN performance

- **How many comparisons does the join make?**

JOIN performance

- **How many comparisons does the join make?**
- Each entity of the first table must be compared.
- For each entity of the first table there is a comparison with each entity of the second for each selection condition.
- **Total comparisons:** $2 \cdot 10 \cdot 3 = 60$.
- **How many operations does JOINING two tables with one condition, respectively with N and M records, require?**

JOIN

JOIN performance

- **How many comparisons does the join make?**
- Each entity of the first table must be compared.
- For each entity of the first table there is a comparison with each entity of the second for each selection condition.
- **Total comparisons:** $2 \cdot 10 \cdot 3 = 60$.
- **How many operations does JOINING two tables with one condition, respectively with N and M records, require?**
- $N \cdot M$.
- **What if there are C conditions?**

JOIN performance

- **How many comparisons does the join make?**
- Each entity of the first table must be compared.
- For each entity of the first table there is a comparison with each entity of the second for each selection condition.
- **Total comparisons:** $2 \cdot 10 \cdot 3 = 60$.
- **How many operations does JOINING two tables with one condition, respectively with N and M records, require?**
- $N \cdot M$.
- **What if there are C conditions?**
- $C \cdot N \cdot M$.

Indexing

Dictionaries and Indices

- A *Dictionary* is a collection of *entries*.
- An *entry* is a pair $\langle key, value \rangle$.
- An index can be thought as a dictionary. The key is the search parameter in the index.

Hash table

- It is an array of *buckets*. Each element of the array is a pointer to a specific bucket.
- A *bucket* is an array used to contain data.
- Insertion uses a *hash function* to map a key to a position in the array.
- The *hash function* always returns a value between 0 and `array.length - 1`.
- Use the hash function to search for a key and go to the corresponding bucket.

Hash table

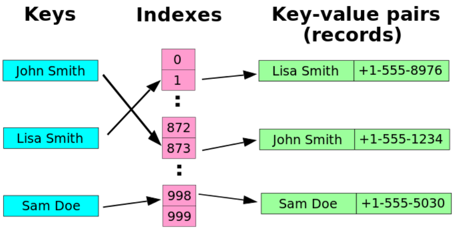


Figure: Example of data insertion in a hash table

Hashing

Collisions

- The hash function might return duplicate values for two different keys, or for duplicate keys.
- We add an element to the bucket for each collision.
- **What happens if the bucket is full?** (*overflow*)

Collisions

- The hash function might return duplicate values for two different keys, or for duplicate keys.
- We add an element to the bucket for each collision.
- **What happens if the bucket is full?** (*overflow*)
- Add a list as overflow bucket. All overflowing entries are stored in the overflow bucket.
- Extend the bucket array when needed.

Performance (WHERE)

- Consider the query on ships presented in the WHERE slides. Suppose we have a hash table on `firepower`.
- **How many operations does the selection require?**

Performance (WHERE)

- Consider the query on ships presented in the WHERE slides. Suppose we have a hash table on `firepower`.
- **How many operations does the selection require?**
- 1 to access the correct bucket, plus 3 operations to read the records in the same bucket = 4 operations vs 10 of non-indexed implementation.

Performance (JOIN)

- **How can the join be implemented?**

Performance (JOIN)

- **How can the join be implemented?**

```
for (p : projectiles)
  for (s : ships)
    if (s.name = p.target &&
          s.position = p.position)
           add <s.name,p.damage> to result
```

- We can index one of the tables and put the indexed query as inner query in the for-loop.

Hashing

Performance (WHERE)

- **What is the cost of the join with a hash index on** `name`[a]?

Performance (WHERE)

- **What is the cost of the join with a hash index on** `name`[a]**?**
- For each record in `projectile` we need to access the bucket in the other table
- The only record which matches in `ship` is `'Red 3'`.
- We apply the hash function and read the entry in the hash table for 'Red 3'. This means $3 \cdot 2 = 6$ operations to select the ships in both table with the same name.
- In total we have $6 + 30$ operations $= 36$ operations (30 for the non-indexed attribute in the condition).

[a] Assume that the index maps the starting letter of its position in the alphabet order and no collisions

Performance (WHERE)

- **What if we also have an index on** `position`**?**

Performance (WHERE)

- **What if we also have an index on** `position`?
- There is one entity in the `ships` table which matches the condition.
- For each record in `projectiles` we look for a record in `ships` with the same position.
- 2 records mismatch the position and one matches.
- Accessing the mismatching records requires 2 operations (we only need to hash the search key to find it is not in the buckets). Accessing the matching record requires 2 operations. In total 4 operations.
- With two indices we require only $6 + 4 = 10$ operations.

Hashing

Drawbacks of hash tables

- Useless if the condition is not an equality but an inequality (conditions on intervals).
- Useless with ordering/grouping commands.
- A lot of collisions require to extend the buckets/slow down the search if overflow buckets are used.

Trees

Balanced trees

- A graph is a set of vertices connected by edges.
- A tree is a graph without cycles and connected (all nodes can be reached following connections from a starting node).
- The top node is called `root`.
- The nodes that are directly connected to a node and at a deeper level are called children. The children shares the same parent.
- A node which does not have children is called *leaf*.
- A balanced tree is a tree where the leaves are all at the same level.

Trees

2-3 Trees

- Balanced tree.
- Every node contains at most 2 entries
- Every node has 2 or 3 children.
- The keys in the left child are less than or equal to the keys in the parent.
- The keys in the middle child are between the min and the max keys stored in the parent.
- The keys in the right child are greater than or equal to the keys in the parent.

2-3 Trees

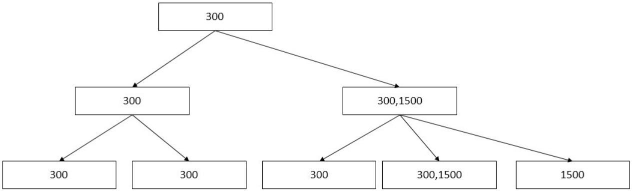


Figure: Index on `firepower`

Performance (WHERE)

- Consider the query on ships presented in the WHERE slides. Suppose we have a 2-3 tree index on `firepower`.
- **How many operations does the selection require?**

Trees

Performance (WHERE)

- Consider the query on ships presented in the WHERE slides. Suppose we have a 2-3 tree index on `firepower`.
- **How many operations does the selection require?**
- We follow the index and we find 3 entries, reading 4 nodes. We need 4 operations vs 10 operations without an index.

Performance (JOIN)

- **What is the cost of the join with a 2-3 tree index on** `name` in table `ships` (see the next slide for the index) [a]?

Performance (JOIN)

- **What is the cost of the join with a 2-3 tree index on** `name` in table `ships` (see the next slide for the index) [a]?
- This time the inner table in the for-loop is `ships`.
- We need to find 'Red 3' three times in the index. This requires to traverse the whole three 3 times and each time we access the record, for a total of 12 operations.
- We have 12 operations vs the required 30 without indexing.
- In total we have 42 operations vs 60 (`position` is not indexed).
- This is a very unlucky case, since the entry is stored in a leaf. Imagine we were looking for "Red 4", we would need just 1 operation to get to the correct node.

[a] Assume that the index maps the starting letter of its position in the

Performance (JOIN)

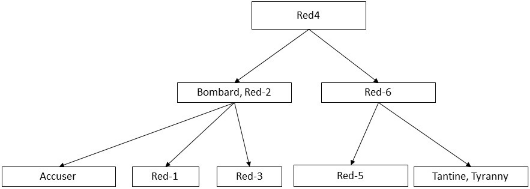


Figure: Index on `name`

Performance (JOIN)

- **Can you guess what is the generalized formula for the number of operations to search for a key in a 2-3 tree?**

Performance (JOIN)

- **Can you guess what is the generalized formula for the number of operations to search for a key in a 2-3 tree?**
- $\log_2 N$ where N is the number of entries.

Performance (JOIN)

- **What if we also have an index on** `position` (see the next slide for the index)[1]?

[1] Imagine that the key is generated by summing the components of the position

Performance (JOIN)

- **What if we also have an index on** `position` (see the next slide for the index)[1]?
- The position will not match for 2 out of 3 records in `projectile`. So we have to traverse 3 nodes in the tree twice for a total of 6 operations.
- The last position will match and the entry is stored in the root so we need just 1 operation.
- In total we have 7 operations.
- With both indices we need 19 vs 60 operations without indices.

[1] Imagine that the key is generated by summing the components of the position

Performance (JOIN)

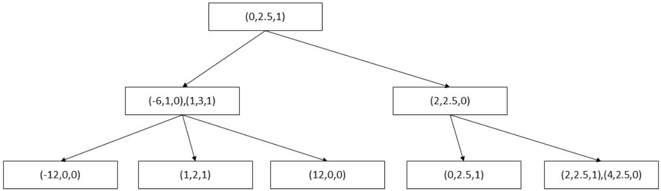


Figure: Index on `position`

Trees

Ordering/grouping with trees

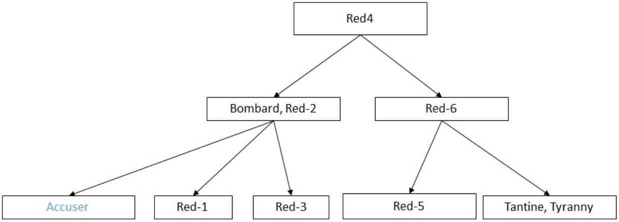
- A clustered index is an index where the record order in a file is the same in the index.
- If the tree index is clustered we can use the tree for ordering/grouping.
- Just traverse the tree. If the ordering is ascending then visit all the nodes starting from the leftmost and then moving to the rightmost.
- This requires N steps where N is the number of entries in the index (vs $N \log N$ of a traditional ordering algorithm).

Ordering/grouping with trees

Exercise: order the `name` attributes in ascending order.

Ordering/grouping with trees

Exercise: order the `name` attributes in ascending order.



Overall evaluation of trees

- Fast with conditions on intervals.
- Fast with ordering/group by.
- The worst case requires to scan $\log N$ entries before finding our entries (with Hash tables it is at most the size of the largest bucket).

Golden rules of indexing

- If most of the queries have selections/joins with equalities choose hash tables.
- If most of the queries have selections/joins with inequalities choose trees.
- If you can anticipate there will be a lot of mismatches in the comparisons choose hash tables.
- If you anticipate there will be a lot of collisions (a lot of duplicate values) choose trees.
- If you have a lot of queries with ordering/group by use clustered trees (otherwise do not use clustering).