DBs: ~~harder~~, better, faster, ~~stronger~~ reading

Maxime Gosselin
Lead Software Architect
Oodrive

MERCREDI 22 JUIN 2016
OPEN R&DAY {}

The big plan

The big plan

PostgreSQL… and the others

The big plan

PostgreSQL… and the others

Storage

The big plan

PostgreSQL… and the others

Storage

Indexes

The big plan

PostgreSQL… and the others

Storage

Indexes

Algorithms, algorithms everywhere!

The big plan

PostgreSQL… and the others

Storage

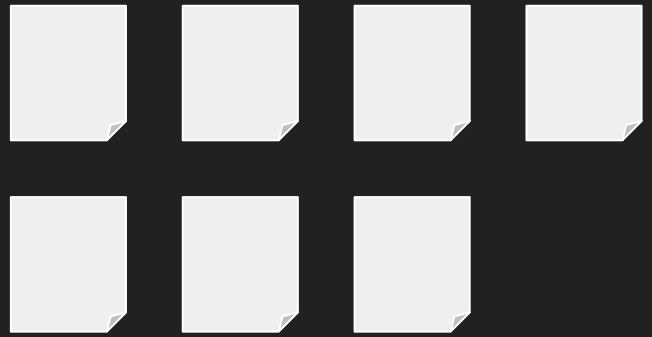
Indexes

Algorithms, algorithms everywhere!

Storage

Pages

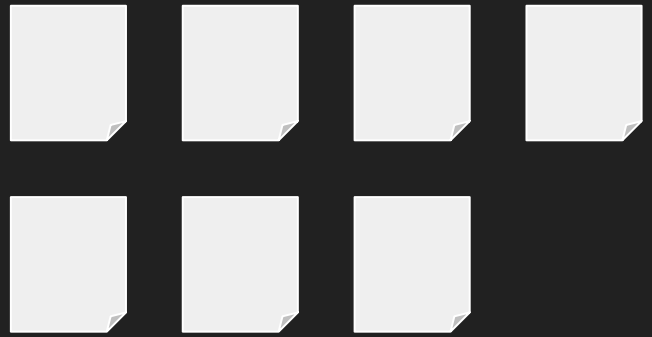
Small, easy to read



Storage

Pages

Small, easy to read



Examples:

Postgresql, Oracle 8kb
MySQL 16kb

Storage

Simple table, ***values***:

- id: integer
- value: text

Storage

Simple table, ***values***:

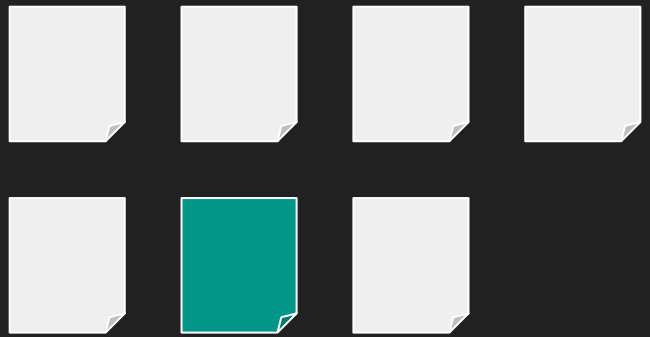
- id: integer
- value: text

Sequential scan

```
SELECT * FROM values WHERE id = 123;
```

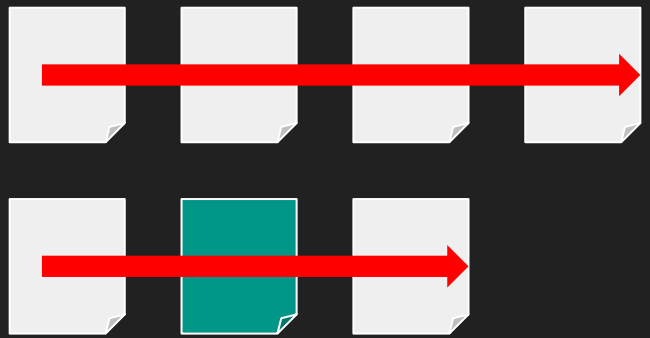
Sequential scan

SELECT * FROM values WHERE id = 123;



Sequential scan

SELECT * FROM values WHERE id = 123;



Every page is read in order.

OK for small tables.

Scales in O(n)

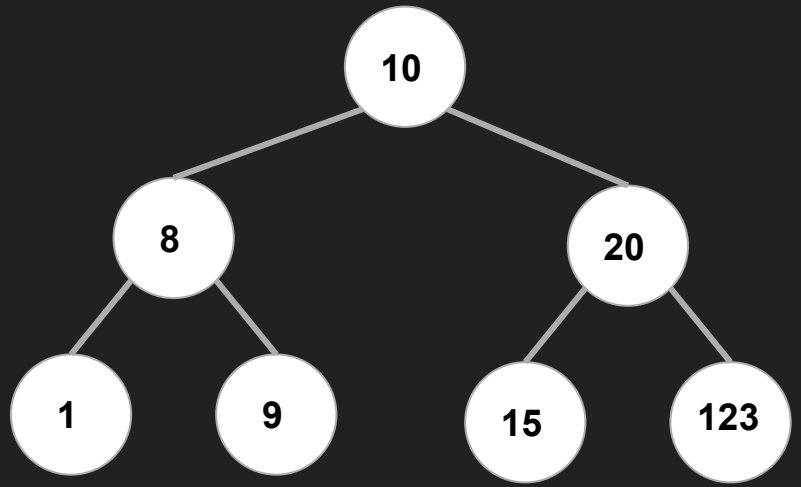
Indexes

Indexes

In the beginning was the binary search tree

Indexes

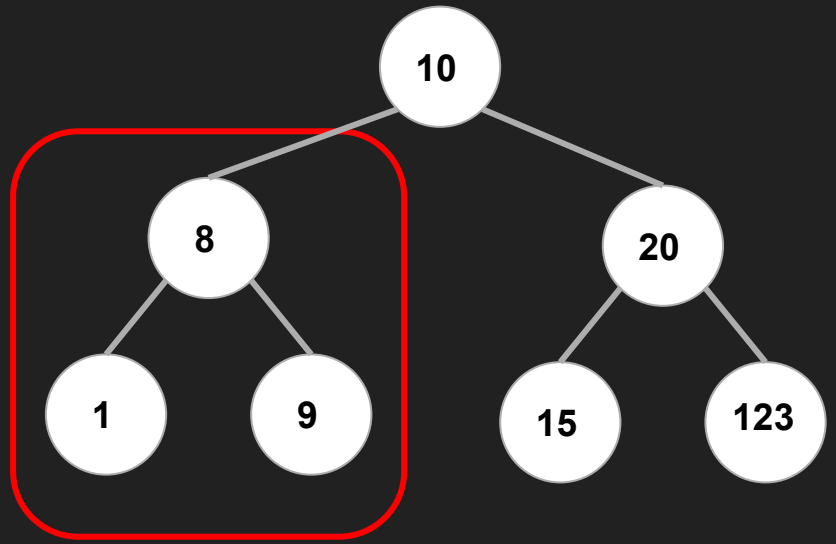
At each node



Indexes

At each node

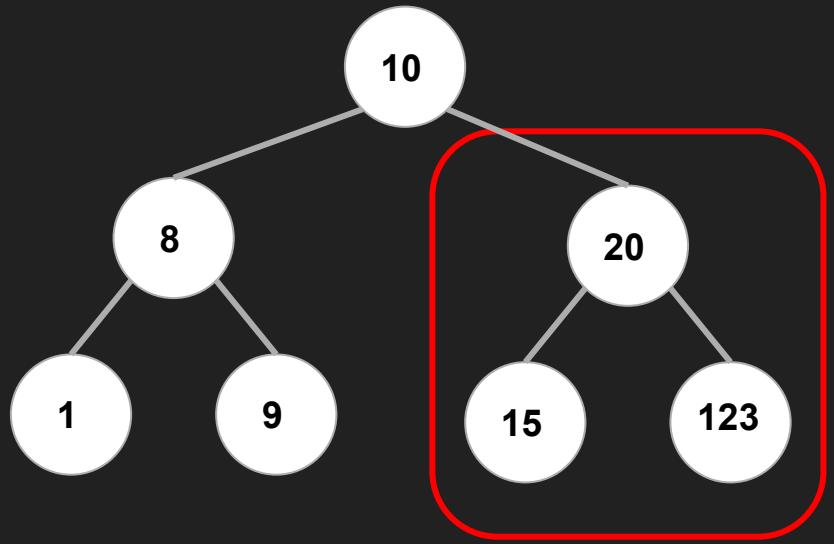
- Left side smaller



Indexes

At each node

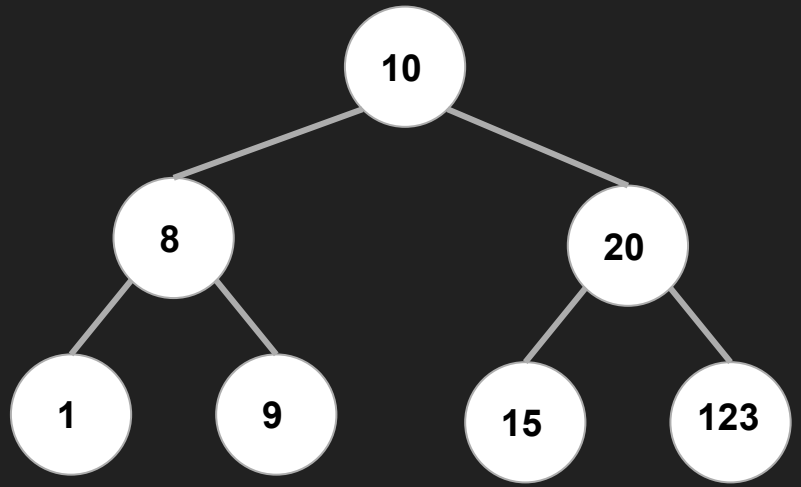
- Left side lower
- Right side higher



Indexes

At each node

- Left side lower
- Right side higher



Not good on disk

Indexes

Let’s complicate things

Indexes

Let’s complicate things

Here comes the B-tree

Indexes

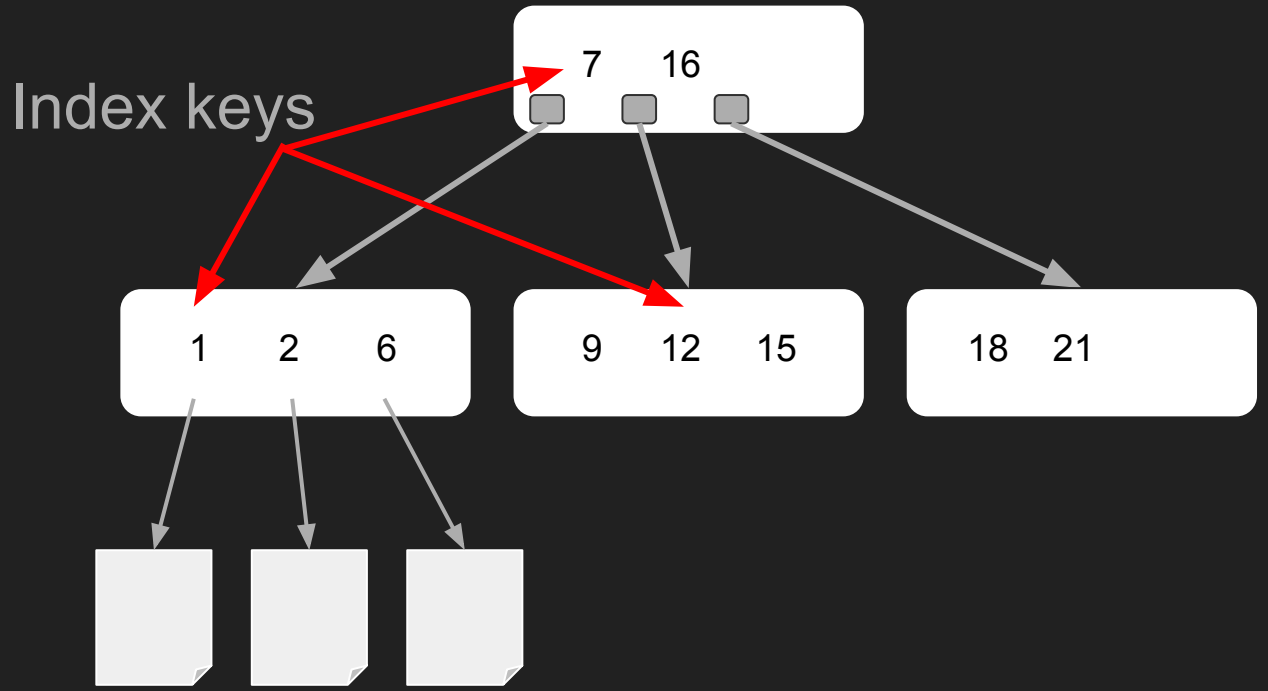
Let’s complicate things

Here comes the B-tree

(Nobody knows what the B means…)

Indexes

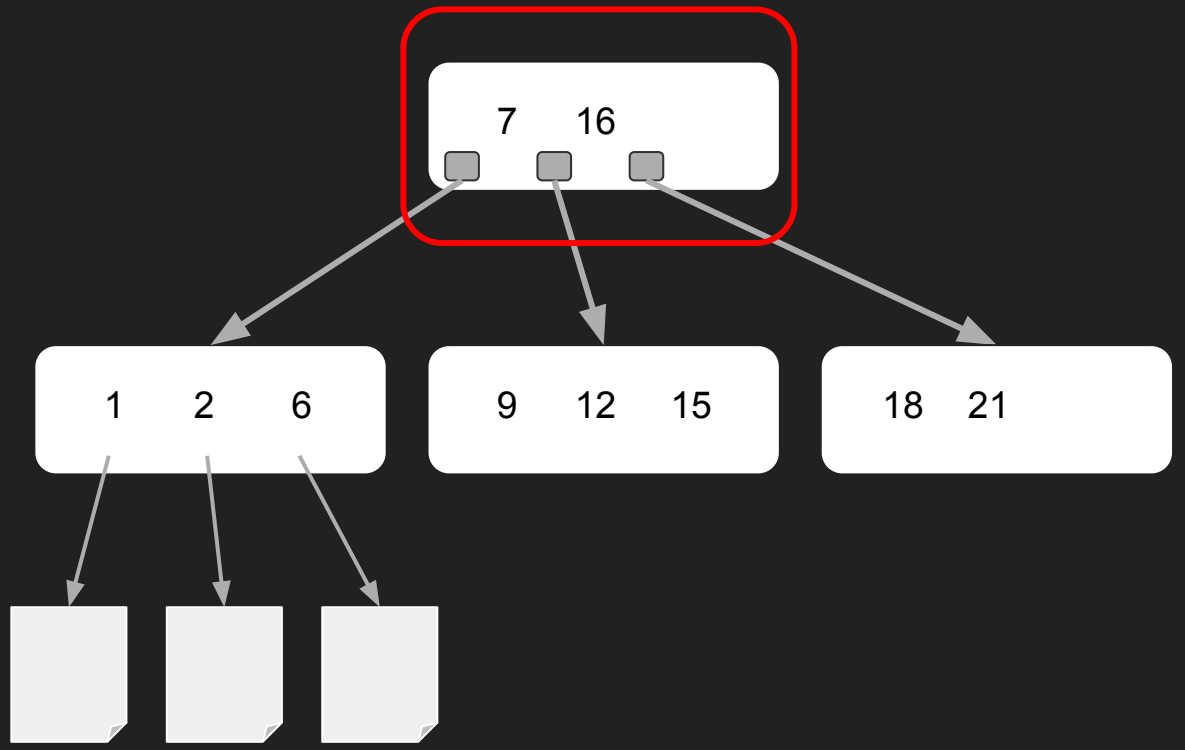
B-tree



Indexes

B-tree

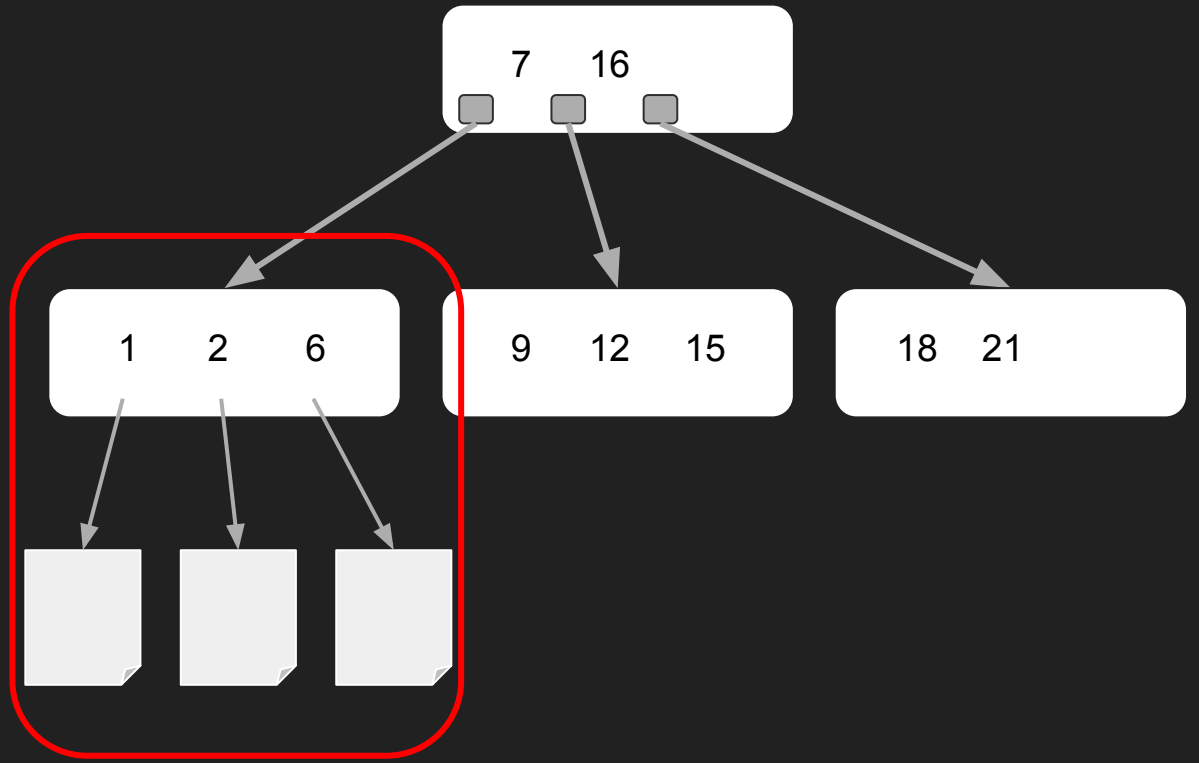
- Internal nodes: Linked to other nodes



Indexes

B-tree

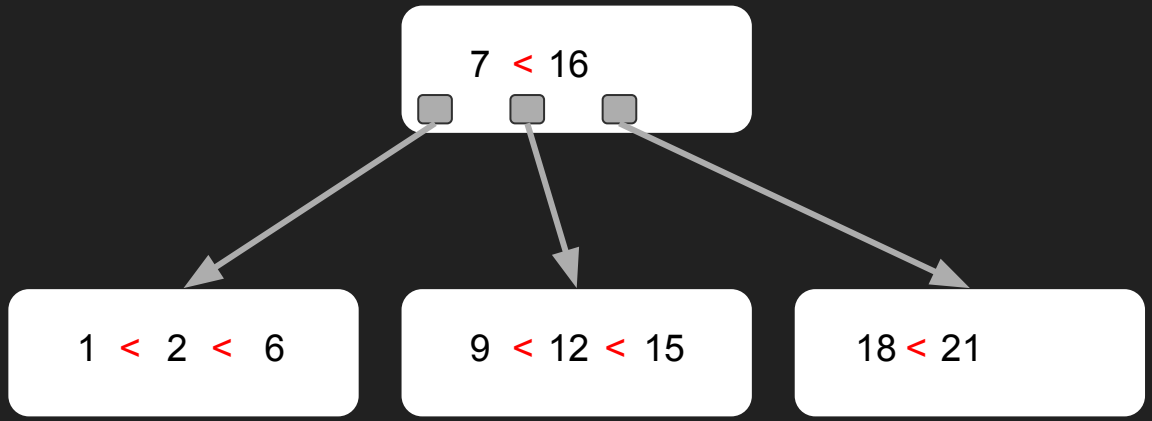
- Internal nodes: linked to other nodes
- Leaves: linked to data pages



Indexes

B-tree

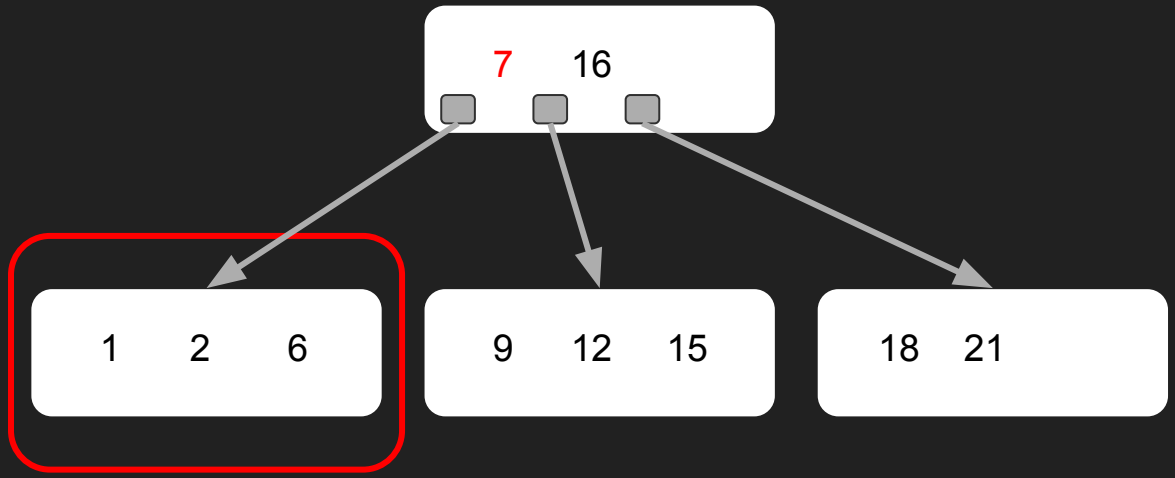
- Internal order



Indexes

B-tree

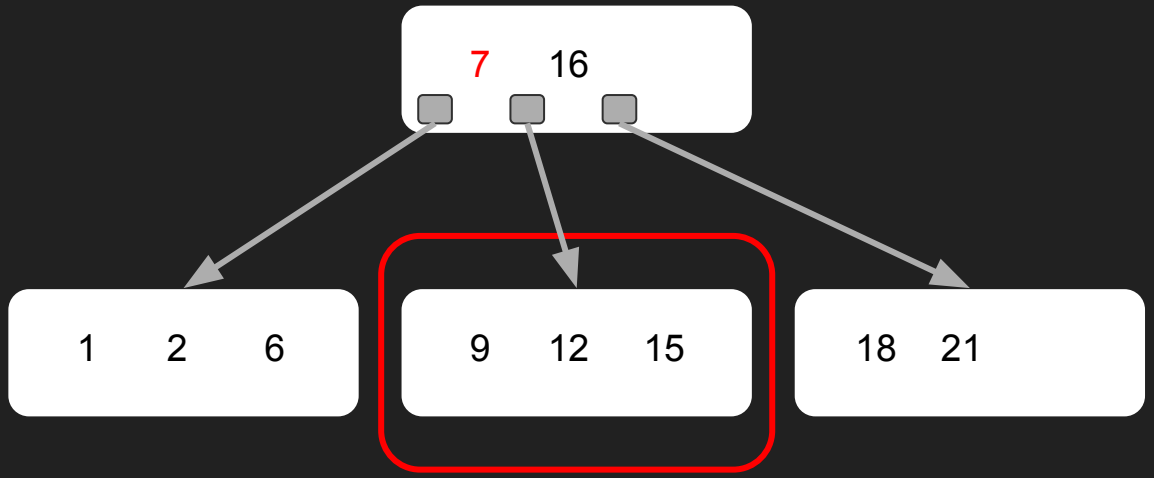
- Internal order
- Left side lower



Indexes

B-tree

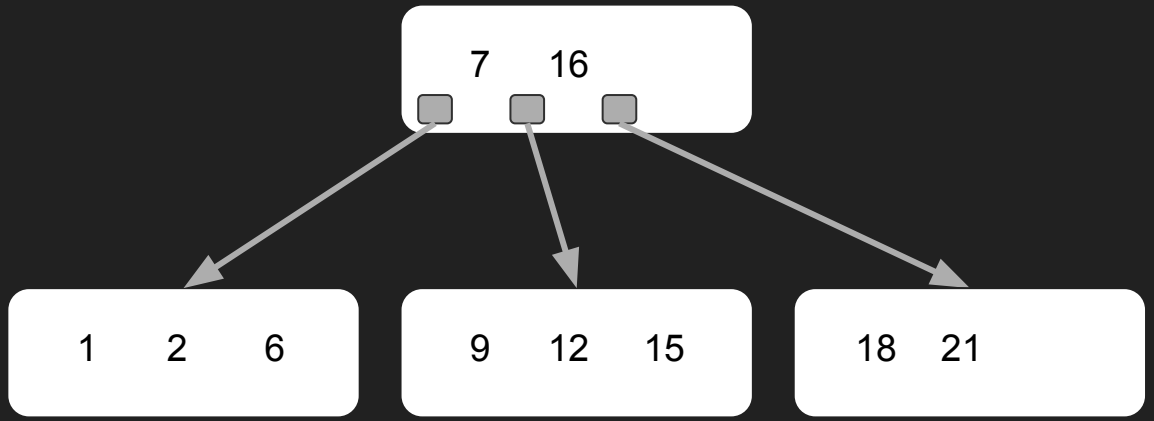
- Internal order
- Left side lower
- Right side higher



Indexes

B-tree

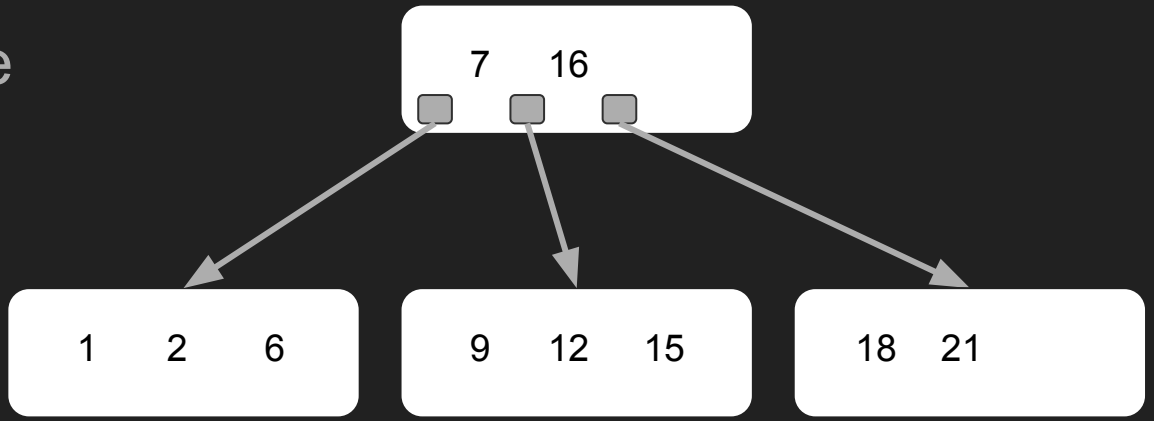
Fits well on a page
=> Efficient on disk



What can we do with B-trees?

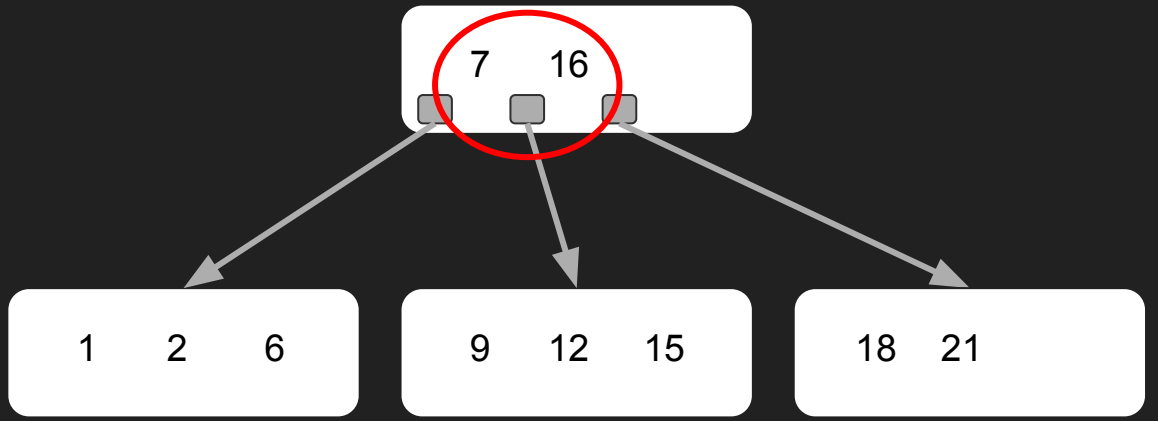
Finding a single value

Let’s look for 12



What can we do with B-trees?

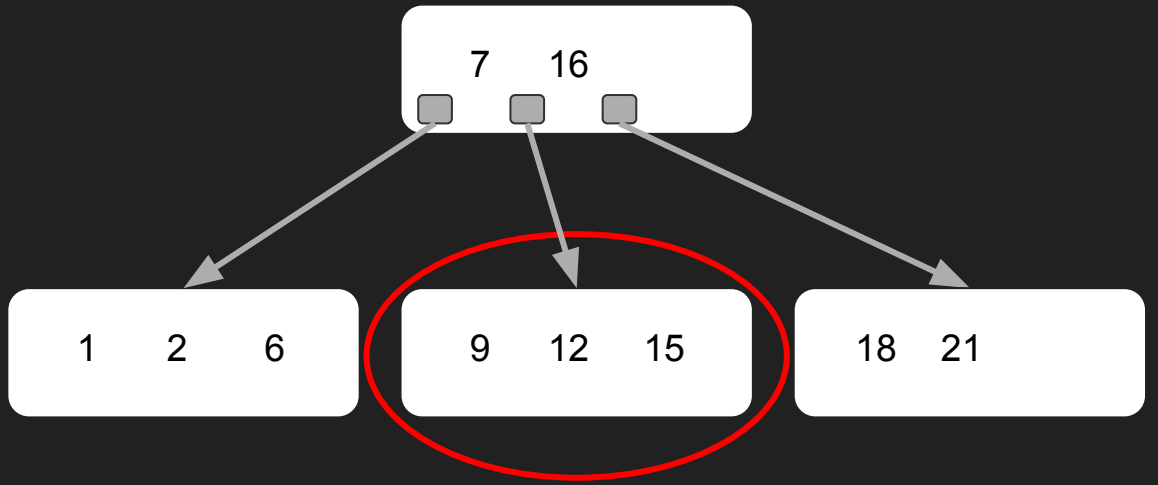
$7 < 12 < 16$



What can we do with B-trees?

7 < 12 < 16

=> follow the pointer

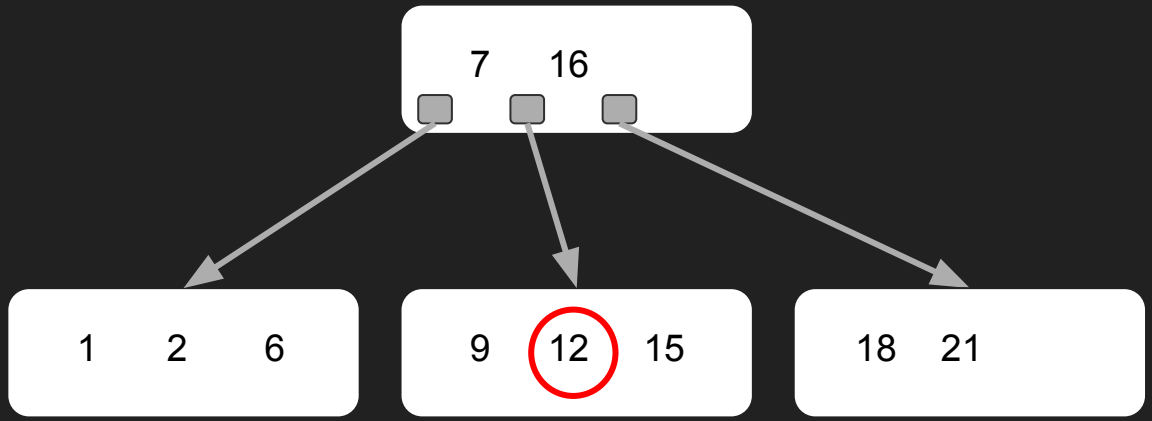


What can we do with B-trees?

7 < 12 < 16

=> follow the pointer

Find it in the page



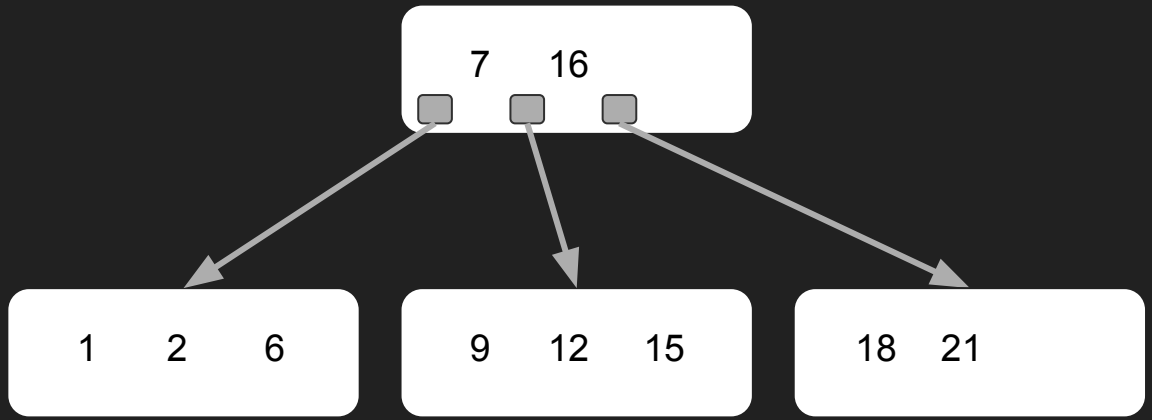
What can we do with B-trees?

7 < 12 < 16

=> follow the pointer

Find it in the page

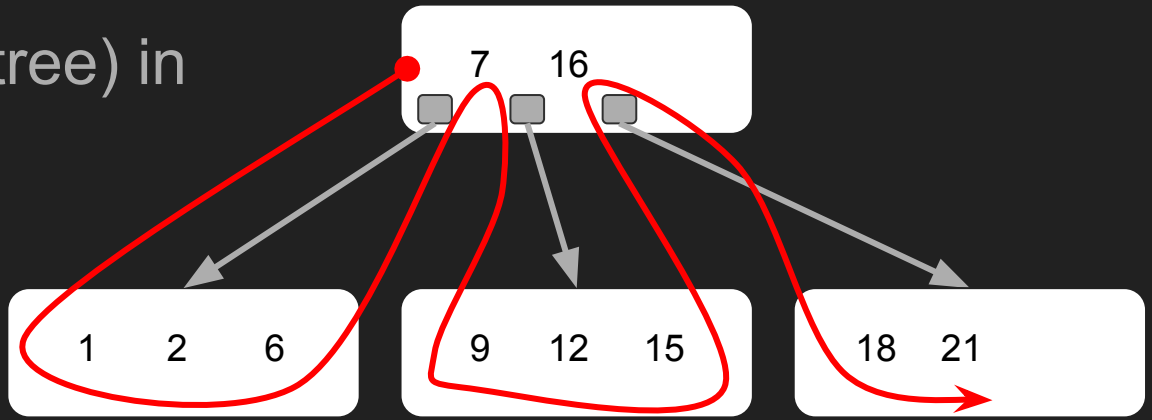
Complexity O(log(n))



What can we do with B-trees?

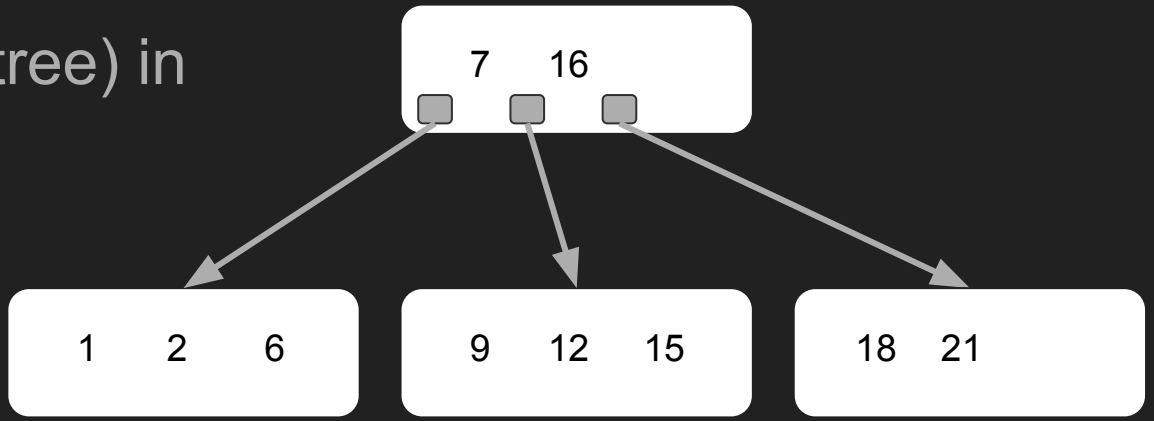
Walk the tree (or subtree) in order

Simple depth first search



What can we do with B-trees?

Walk the tree (or subtree) in order



=> Answer ORDER BY queries

What CAN’T we do with B-trees?

Answer queries with:

- AND clauses using multiple trees

What CAN’T we do with B-trees?

Answer queries with:

- AND clauses using multiple trees

SELECT * FROM table WHERE col1 = 123 AND col2 = ‘abc’;

With an index on col1 and another on col2.

What CAN'T we do with B-trees?

Answer queries with:

- AND clauses using multiple trees

SELECT * FROM table WHERE col1 = 123 AND col2 = 'abc';

With an index on col1 and another on col2.

But a single index on (col1, col2) would work nicely.

What CAN’T we do with B-trees?

Answer queries with:

- clauses on values derived from an indexed column

SELECT * FROM people WHERE lower(name) = ‘maxime’;

What CAN’T we do with B-trees?

Answer queries with:

- clauses on values derived from an indexed column

SELECT * FROM people WHERE lower(name) = ‘maxime’;

But you can build an index on (lower(name))

Multi column indexes

Column order matters!

(Company, city)

Blablacar,	Paris
Canal+,	Issy les Moulineaux
Oodrive,	Lyon
Oodrive,	Paris
...	

Multi column indexes

Column order matters!

(Company, city)

Graphs are too hard.
But it's still a B-tree!

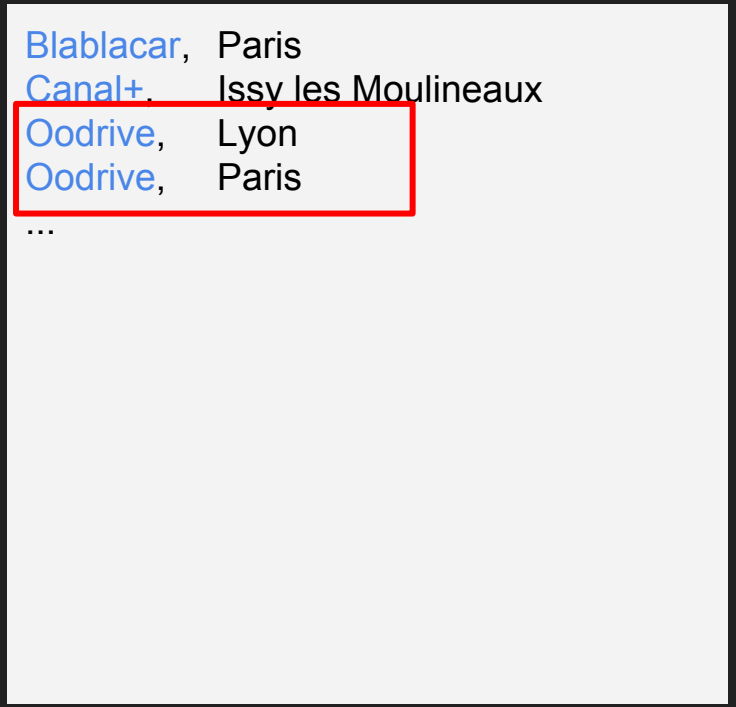
Blablacar,	Paris
Canal+,	Issy les Moulineaux
Oodrive,	Lyon
Oodrive,	Paris
...	

Multi column indexes

Column order matters!

(Company, city)

It’s easy to find all the cities for Oodrive

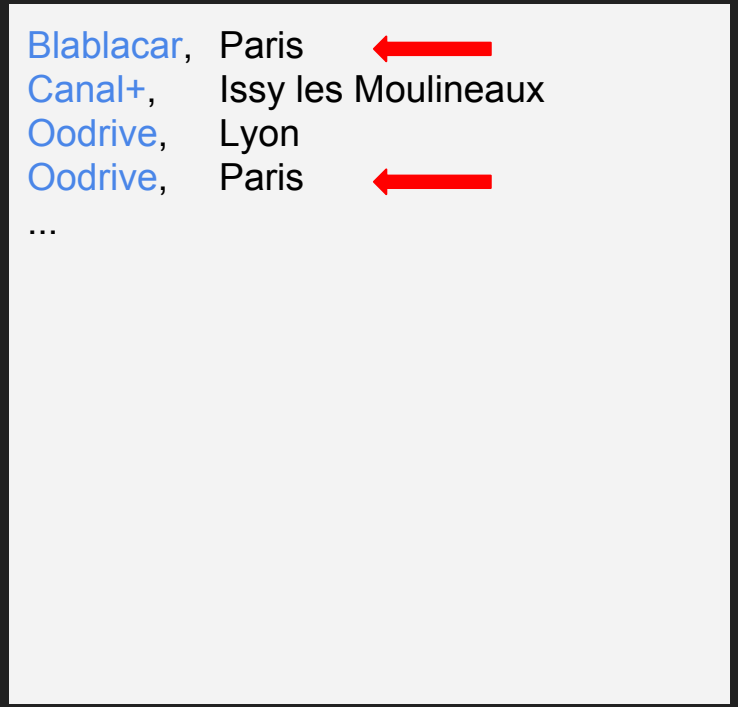


Multi column indexes

Column order matters!

(Company, city)

It’s hard to find who works in Paris

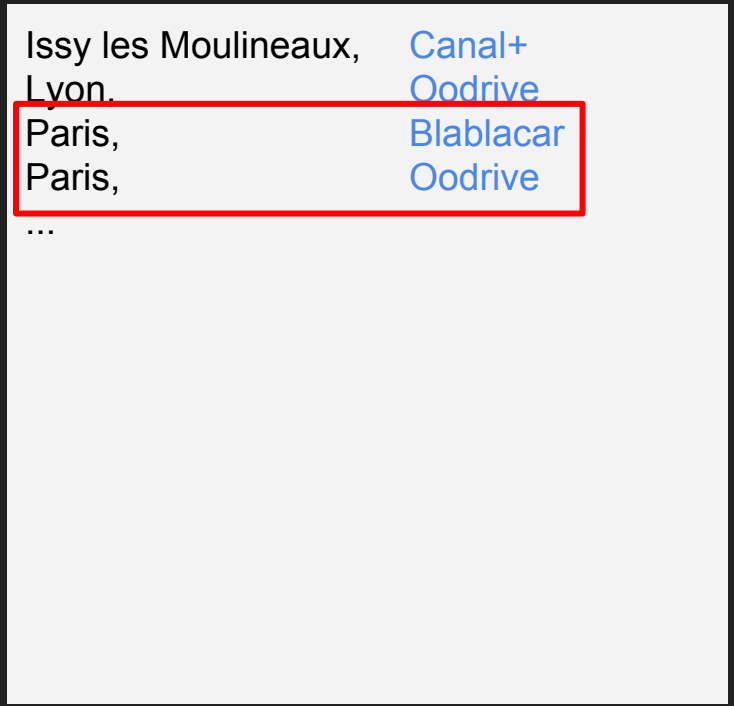


Multi column indexes

Column order matters!

(City, Company)

It’s easy to find who works in Paris

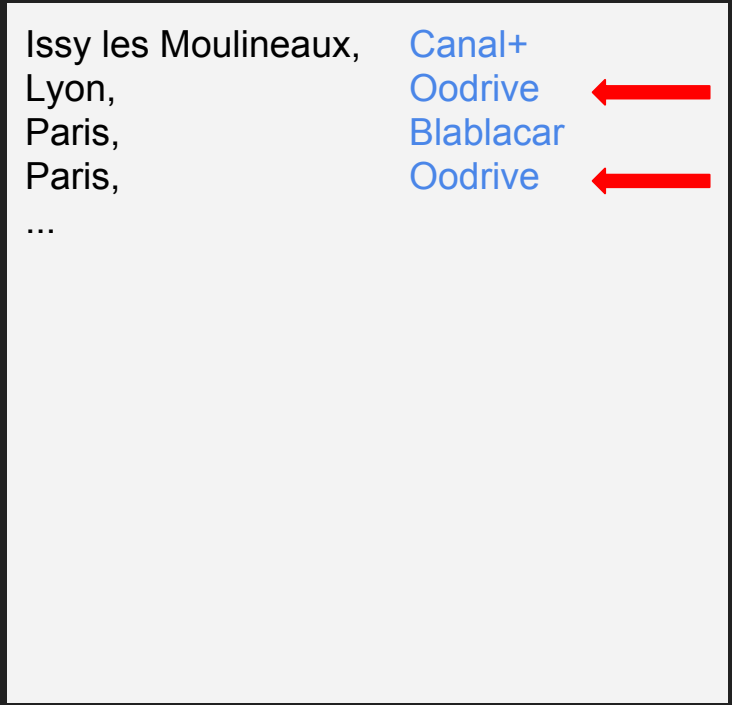


Multi column indexes

Column order matters!

(City, Company)

It’s hard to find all the cities for Oodrive



Joining tables

Joining tables

```
SELECT * FROM table1
        JOIN table2 USING (joinColumn1)

    WHERE ...;
```

Joining tables: nested loop

Standard developper approach:

```
for(int i = 0; i < table1.length; ++i) {
      filter(table1[i]); ….
      for(int j = 0; j < table2.length; ++j) {
            filter(table2[j]); ….
      }
}
```

Joining tables: nested loop

Works well if:

- the first table is small,
- the second table has an index on the join column.

Joining tables: nested loop

Works well if:

- the first table is small,
- the second table has an index on the join column.

Doesn’t if:

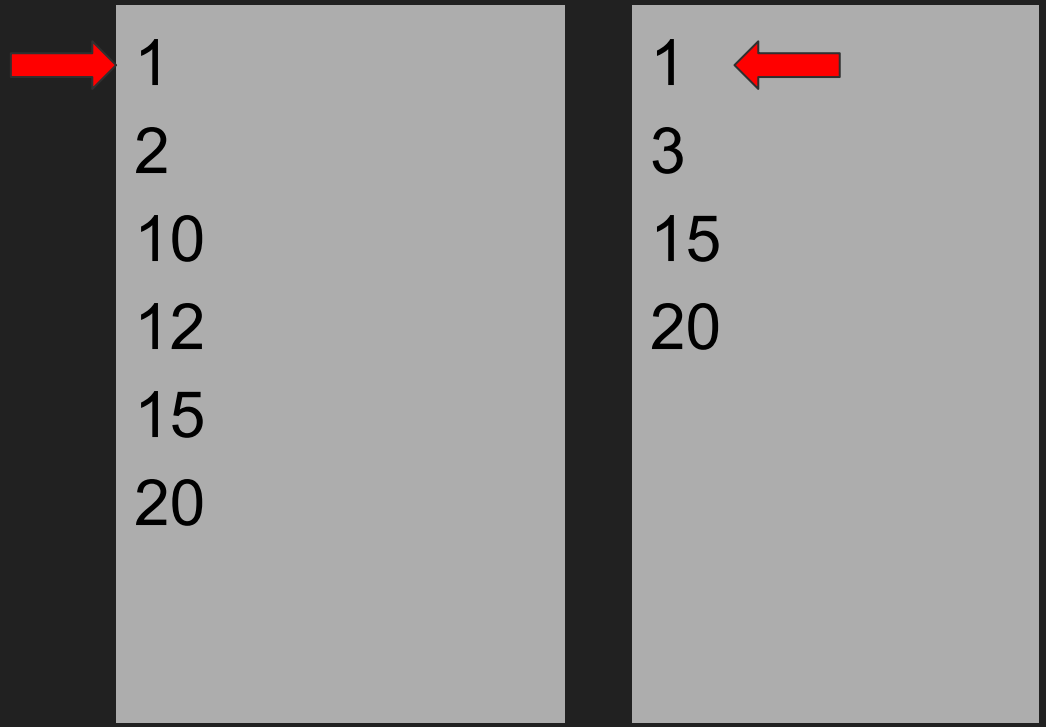
- the join condition is complex

Joining tables: merge join

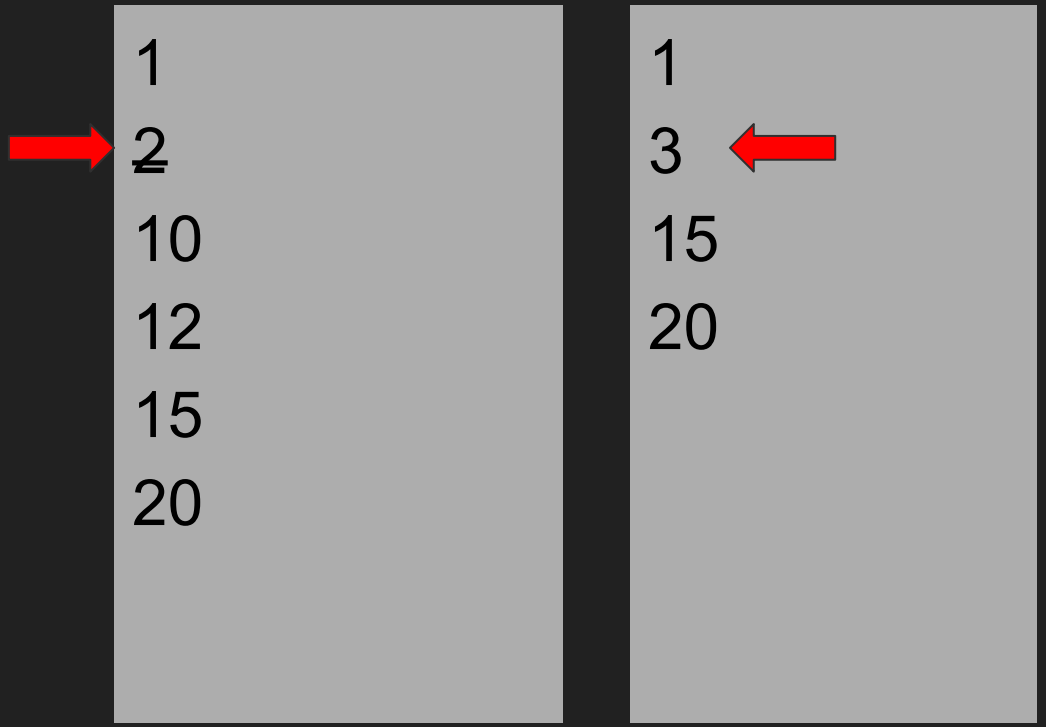
Sort the two sides of the join on the index condition

Iterate both sides to find common values.

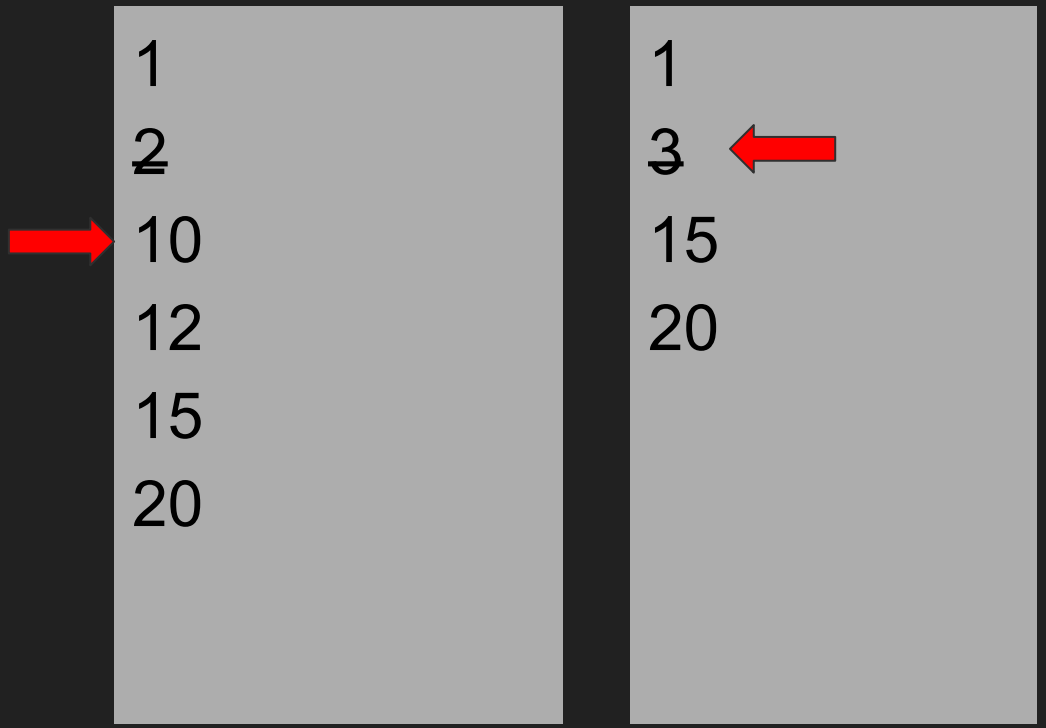
Joining tables: merge join



Joining tables: merge join



Joining tables: merge join



Joining tables: merge join

Works well if:

- both sides can be filtered and sorted cheaply
- both sides are big

Best strategy for full outer joins

Joining tables: hash join

On the smaller side:

- filter the data
- put each row in a hash table, indexed by the join columns

Joining tables: hash join

```
SELECT * FROM t1
    JOIN t2 using(a, b);
```

Row1: a1, b1, c1, ...

Row2: a2, b2, c2, ...

Joining tables: hash join

```
SELECT * FROM t1
    JOIN t2 using(a, b);
```

row1: a1, b1, c1, ...

row2: a2, b2, c2, ...

hash(a1, b1) => row1, rowN
hash(a2, b2) => row2
….

Joining tables: hash join

On the smaller side:

- filter the data
- put each row in a hash table, indexed by the join columns

On the larger side

- filter the data
- for every row, retrieve the smaller side from the hash table

Joining tables: hash join

```
SELECT * FROM t1
    JOIN t2 using(a, b);
```

row1: a1, b1, c1, ...

row2: a2, b2, c2, ...

hash(a1, b1) => row1, rowN
hash(a2, b2) => row2
….

Joining tables: hash join

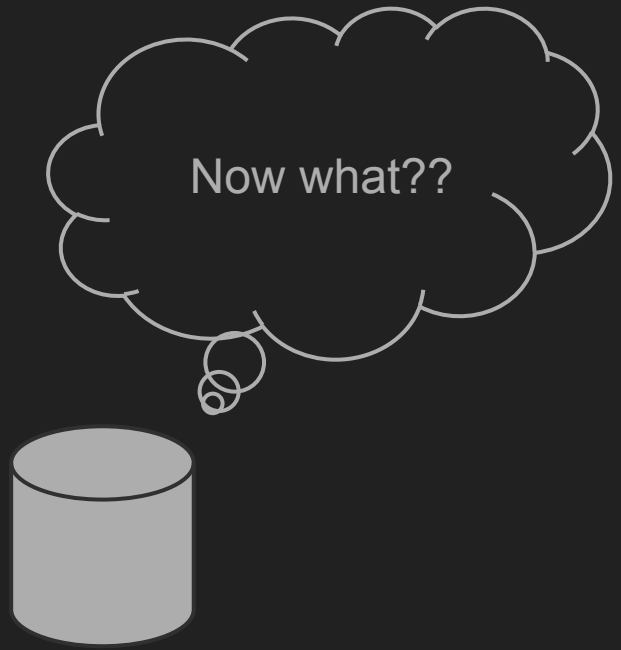
Advantages:

- can work without indexes on the join columns
- single traversal of the smaller side

Drawbacks

- the hash table should fit in memory

Planning & optimizing



Planning & optimizing

Goal: least possible work

=> read as little as possible

Planning & optimizing

The tools:

- a query tree that can be modified
- cardinality estimation for column values

Planning & optimizing

How can it be used?

- generate all possible execution plans (what index to use, join types and order...)
- compute the costs of each plan

Planning & optimizing

How can it be used?

- generate all possible execution plans (what index to use, join types and order...)
- compute the costs of each plan

In the end, there can be only one.

Planning & optimizing

```
Hash Join  (cost=14.25..246593.00 rows=1338306 width=15)
  Hash Cond: (series.val = even.val)
  ->  Seq Scan on series  (cost=0.00..186311.50
rows=12502450 width=12)
  ->  Hash  (cost=8.00..8.00 rows=500 width=7)
      ->  Seq Scan on even  (cost=0.00..8.00 rows=500
width=7)
```

Planning & optimizing

Startup cost

Total cost

Rows expected

```
Hash Join  (cost=14.25..246593.00 rows=1338306 width=15)
  Hash Cond: (series.val = even.val)
  ->  Seq Scan on series  (cost=0.00..186311.50
rows=12502450 width=12)
  ->  Hash  (cost=8.00..8.00 rows=500 width=7)
      ->  Seq Scan on even  (cost=0.00..8.00 rows=500
width=7)
```

Caveats

Let’s index everything!

Caveats

~~Let's index everything!~~

Index only what you need, they are costly

Caveats

~~Let's index everything!~~

Index only what you need, indexes are costly

Small dataset, small gains

Other interesting things

- data modification & MVCC
  (Multiversion concurrency control)

How are transactions represented?

How do we keep uncommitted writes from being read?

Other interesting things

- data modification & MVCC
  (Multiversion concurrency control)
- replication

What are the tradeoffs between consistency and availability?

Other interesting things

- data modification & MVCC (Multiversion concurrency control)
- replication
- other index types

Inverted indexes, bitmap, hash, spatial...

The end

Oh, you had questions?