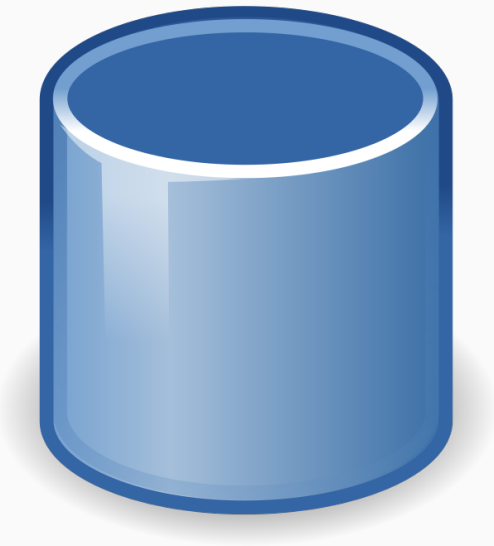
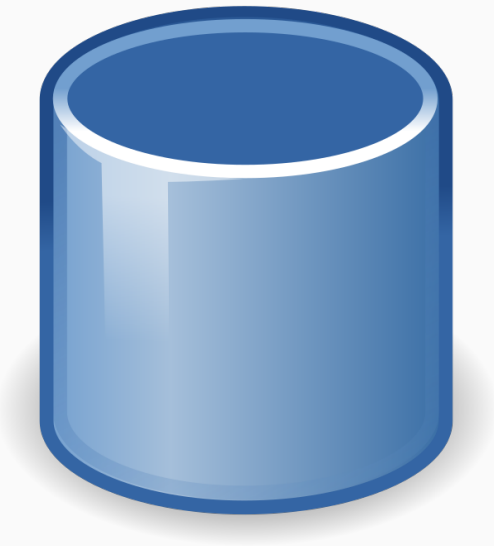
SD202: Evaluation algorithms

Louis Jachiet

Introduction



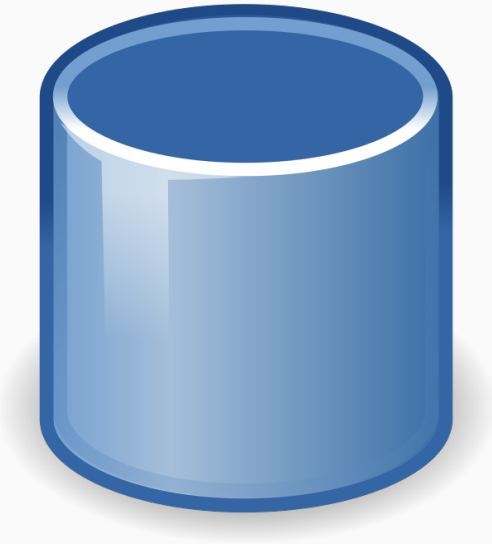
Introduction



Query

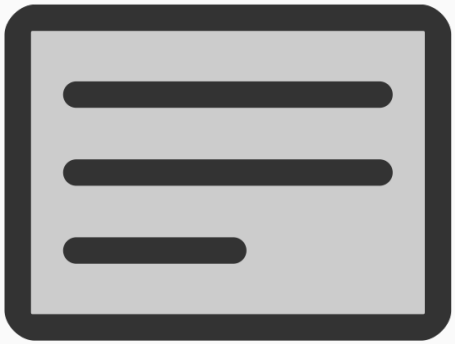
$\Rightarrow$

Introduction

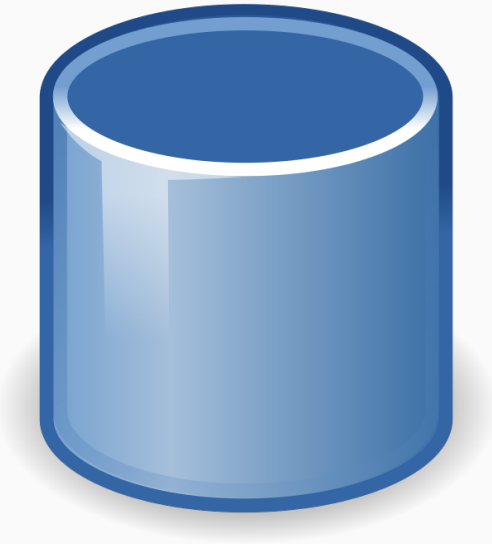


Query

$\Rightarrow$

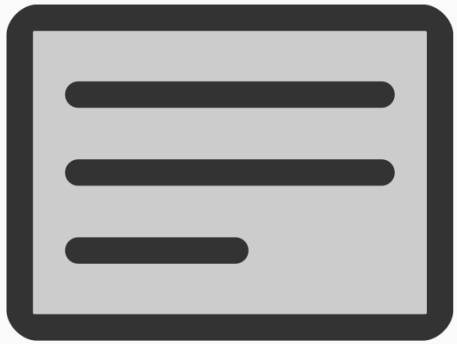


Introduction



Query

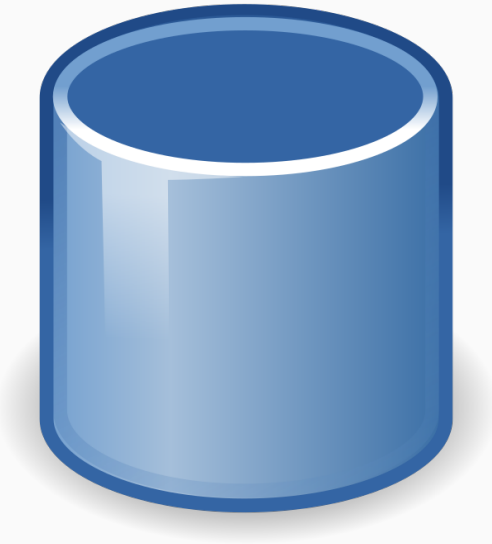
$\Rightarrow$



How to compute

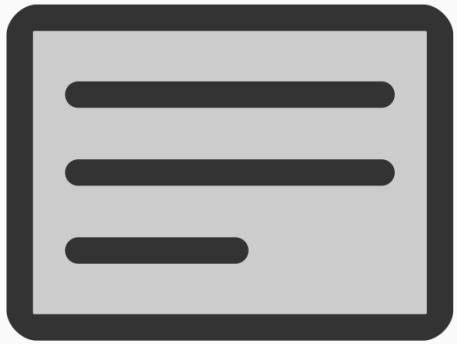
the result of a query?

Introduction



Query

$\Rightarrow$



How to compute efficiently

the result of a query?

Remember the relational algebra?

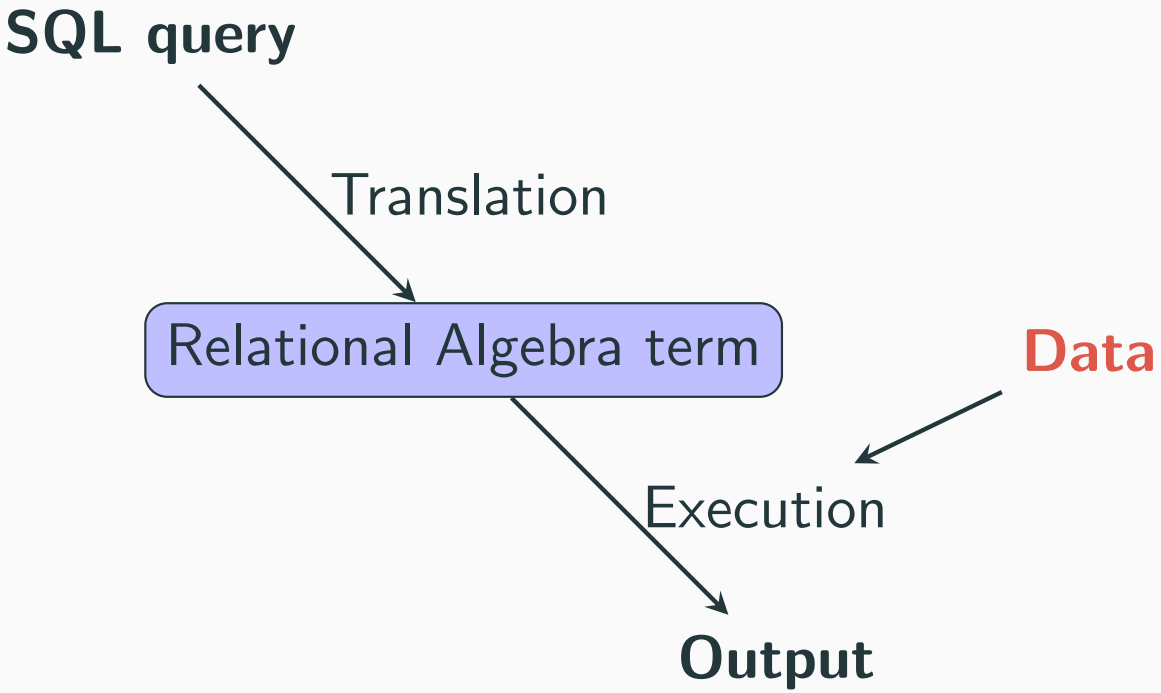
A dialect comprising:

- Relation $\mathcal{R}$
- RENAME(t,a,b)
- DROP(t,a)
- FILTER(t,cond)
- PRODUCT(t,t')
- UNION

And also:

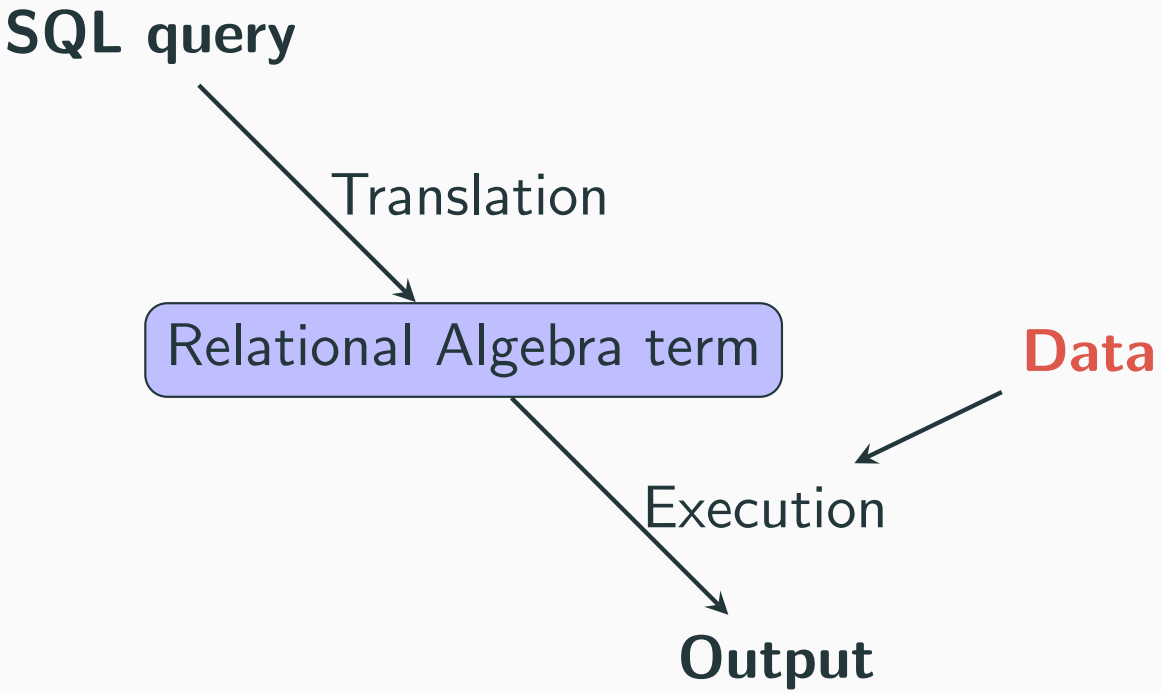
- JOIN(t,t',cond,type)
- AGGREGATE(t,cols, exprs)
- and other operators

Query evaluation in a nutshell



Is it that simple?

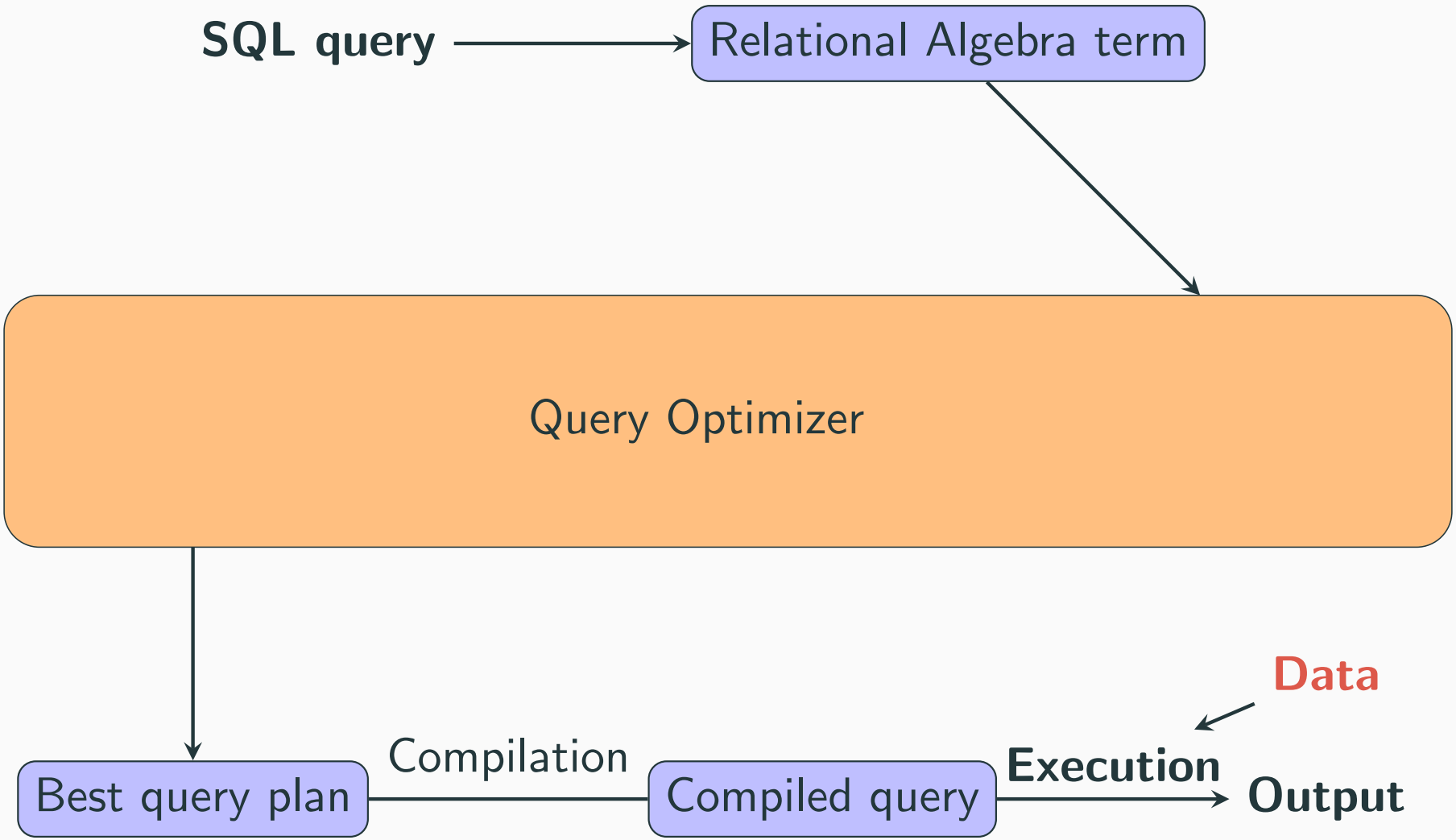
Query evaluation in a nutshell



Is it that simple?

Of course, no :)

Query evaluation in a nutshell



Today's lesson

We will cover several topics

- Logical optimization
- A model of computation time
- Efficient retrieval using indexes
- Several algorithms to joins
- A general method for query optimization
- Some advanced topics (e.g. transactions)

Logical optimization of queries

Comparing relational algebra terms

Consider the two following terms, are they computing the same thing? Is one more efficient than the other?

```
FILTER(
 JOIN(Room, Movie),
 title="Le Bonheur"
)
```

```
JOIN(Room,
  FILTER(Movie,
  title="Le Bonheur")
)
```

Here `JOIN` is the natural join (i.e. on shared attributes)

Comparing relational algebra terms

What about those two terms?

```
JOIN(
  JOIN(Room, Movie),
  OscarMovies
)
```

```
JOIN(
  Room,
  JOIN(OscarMovies,Movie)
)
```

Equivalence rule

We can devise a set of rewriting rules

- `JOIN(a,JOIN(b,c))` $\rightarrow$ `JOIN(JOIN(a,b),c)`

  Associativity

- `JOIN(a,b)` $\rightarrow$ `JOIN(b,a)`

  Commutativity

- `FILTER(JOIN(a,b),f)` $\rightarrow$ `JOIN(FILTER(a,f),b)`

  Distributivity when f *operates on attributes in* a

- And many other rules...

A RA term optimizer

Algorithm

- Start with an RA term φ
- Set oldplans $= \emptyset$, plans $= \{\varphi\}$
- While oldplans $\neq$ plans
  - oldplans $=$ plans
  - plans $=$ plans $\cup \{\varphi' \mid$ where φ' is a rewriting of $\varphi \in$ plans$\}$
- Return the most efficient $\varphi \in$ plans

A RA term optimizer

Algorithm

- Start with an RA term φ
- Set oldplans $= \emptyset$, plans $= \{\varphi\}$
- While oldplans $\neq$ plans
  - oldplans $=$ plans
  - plans $=$ plans $\cup \{\varphi' \mid$ where φ' is a rewriting of $\varphi \in$ plans$\}$
- Return the most efficient $\varphi \in$ plans

In practice, exploring the full plan space is prohibitively expensive for large queries.

A RA term optimizer

Algorithm

- Start with an RA term φ
- Set oldplans $= \emptyset$, plans $= \{\varphi\}$
- While oldplans $\neq$ plans
  - oldplans $=$ plans
  - plans $=$ plans $\cup \{\varphi' \mid$ where φ' is a rewriting of $\varphi \in$ plans$\}$
- Return the most efficient $\varphi \in$ plans

In practice, exploring the full plan space is prohibitively expensive for large queries.

Also, how can we determine the most efficient φ?

Cost model on RA terms

Determining the efficiency of an RA term φ depends on:

- a cost-model for each RA operator
- a cardinality estimation for each subterm appearing in φ

Cost model on RA terms

Determining the efficiency of an RA term φ depends on:

- a cost-model for each RA operator
- a cardinality estimation for each subterm appearing in φ

This simple model ignores the fact one RA operator (e.g. JOIN) can be performed with different algorithms... more on that later!

A cost model to estimate computation time

Modeling computation time

Basic model

Count the number of operations.

Modeling computation time

Basic model

Count the number of operations.

We know it is correct (up to a constant)

Modeling computation time

Basic model

Count the number of operations.

We know it is correct (up to a constant)

Data access time	
CPU L1 cache	$\leq$1ns
CPU L2 cache	5ns
RAM access	0.1μs
SSD seek	0.1ms
HDD seek	10ms

I/O speed (vague numbers!)	
CPU caches	50GB/s
RAM	10GB/s
SSD	500MB/s
HDD sequential	100MB/s
HDD random	30MB/s

Modeling computation time

Basic model

Count the number of operations.

We know it is correct (up to a constant)

Data access time	
CPU L1 cache	$\leq$1ns
CPU L2 cache	5ns
RAM access	0.1μs
SSD seek	0.1ms
HDD seek	10ms

I/O speed (vague numbers!)	
CPU caches	50GB/s
RAM	10GB/s
SSD	500MB/s
HDD sequential	100MB/s
HDD random	30MB/s

Looking only up to a constant factor does not give the full picture!

A more precise model of time

Count the different kinds of operations separately:

- number of seeks $\times$ t_{seek}
- number of blocks read sequentially $\times$ t_{read}
- number of blocks written $\times$ t_{write}
- number of CPU operations $\times$ t_{cpu}
- size of block s_{block} (typically a few kB).

Simple query evaluation

SELECT * FROM myTable

Not much can be done to optimize this FULL SCAN (N sequential reads).

Simple query evaluation

SELECT * FROM myTable

Not much can be done to optimize this FULL SCAN (N sequential reads).

SELECT * FROM myTable
WHERE field="value"

Can we do better than a full scan?

Simple query evaluation

SELECT * FROM myTable

Not much can be done to optimize this FULL SCAN (N sequential reads).

SELECT * FROM myTable
WHERE field="value"

Can we do better than a full scan? It depends!

SELECT * FROM myTable WHERE field=”value”

Algorithm 1

Read the table sequentially.

SELECT * FROM myTable WHERE field="value"

Algorithm 1

Read the table sequentially.

Algorithm 2

Precompute list L with the position of relevant records. For each element of L, retrieve it.

SELECT * FROM myTable WHERE field="value"

Algorithm 1

Read the table sequentially. $\Rightarrow$ N sequential reads.

Algorithm 2

Precompute list L with the position of relevant records. For each element of L, retrieve it.

SELECT * FROM myTable WHERE field="value"

Algorithm 1

Read the table sequentially. $\Rightarrow$ N sequential reads.

Algorithm 2

Precompute list L with the position of relevant records. For each element of L, retrieve it. $\Rightarrow$ L non sequential reads.

SELECT * FROM myTable WHERE field="value"

Algorithm 1

Read the table sequentially. $\Rightarrow$ N sequential reads.

Algorithm 2

Precompute list L with the position of relevant records. For each element of L, retrieve it. $\Rightarrow$ L non sequential reads.

Algorithm 2 is better only when the condition is very selective!

How to get the estimation for t_{seek}, t_{write}, t_{read}, t_{cpu}, etc.

- use default values
- use default values based on hardware
- detect values at boot
- learn values
- ...

How to get the estimation for t_{seek}, t_{write}, t_{read}, t_{cpu}, etc.

- use default values
- use default values based on hardware
- detect values at boot
- learn values
- ...

Default values are generally OK but can often be tuned for optimal performance.

Indexes

Simple query evaluation

SELECT * FROM myTable WHERE field="value"

How to do better than a full scan?

Simple query evaluation

SELECT * FROM myTable WHERE field=”value”

How to do better than a full scan?

With indexes!

Simple query evaluation

SELECT * FROM myTable WHERE field=”value”

How to do better than a full scan?

With indexes!

Different types of indexes:

- Hash
- B-tree
- GIN
- GiST
- ...

Simple query evaluation

SELECT * FROM myTable WHERE field="value"

How to do better than a full scan?

With indexes!

Different types of indexes:

- Hash *for equality tests only*
- B-tree *for arbitrary comparisons*
- GIN *for case-insensitive search, n-grams, prefixes, JSON field lookup, etc.*
- GiST *for non ordered data (e.g. geographical data, date intervals, etc.)*
- ...

Hash indexes

Data

An array A of size K containing lists of pair (key,pointer).

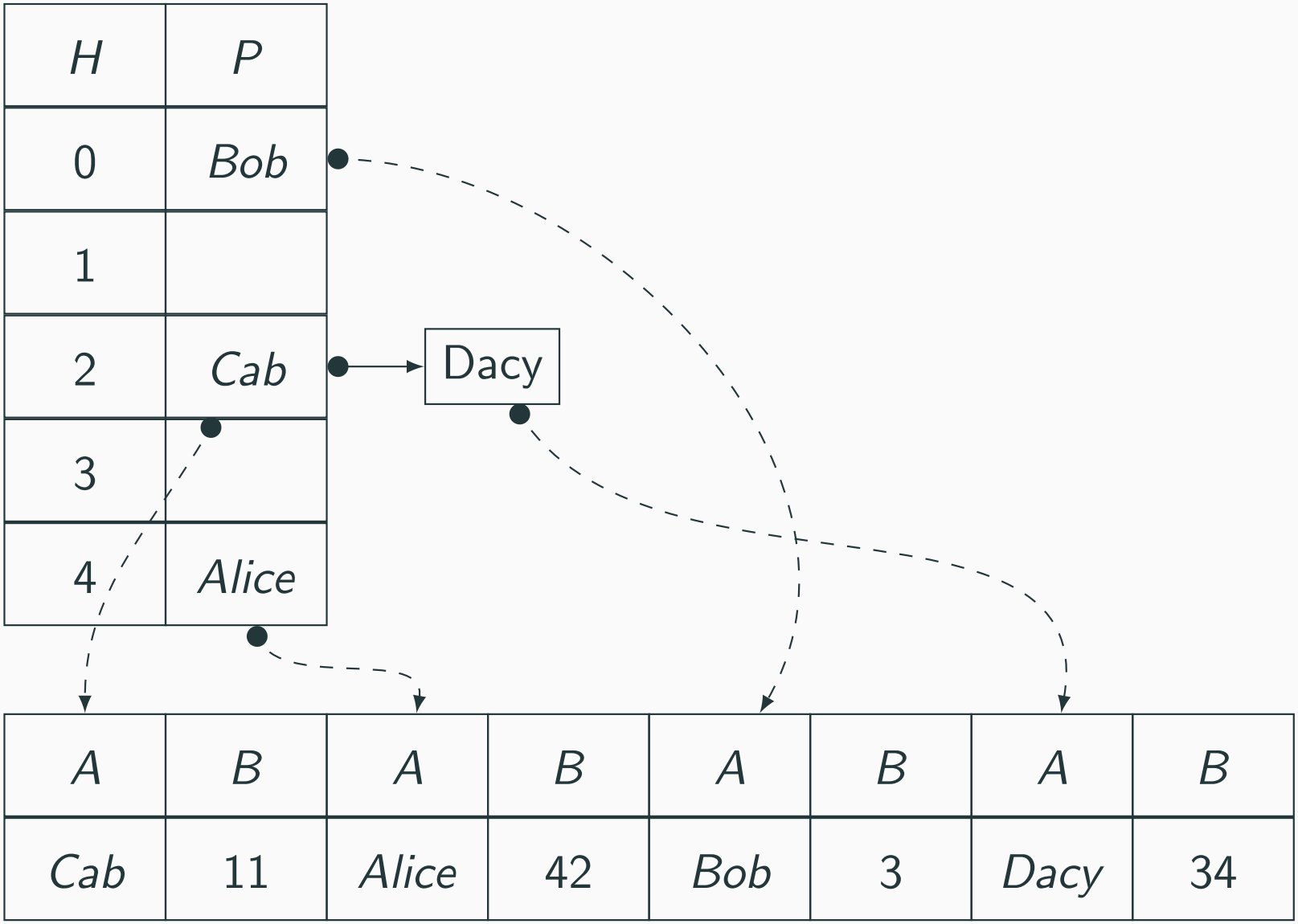
Check key x

Compute hash(x) and iterate over the elements of A(hash(x)).

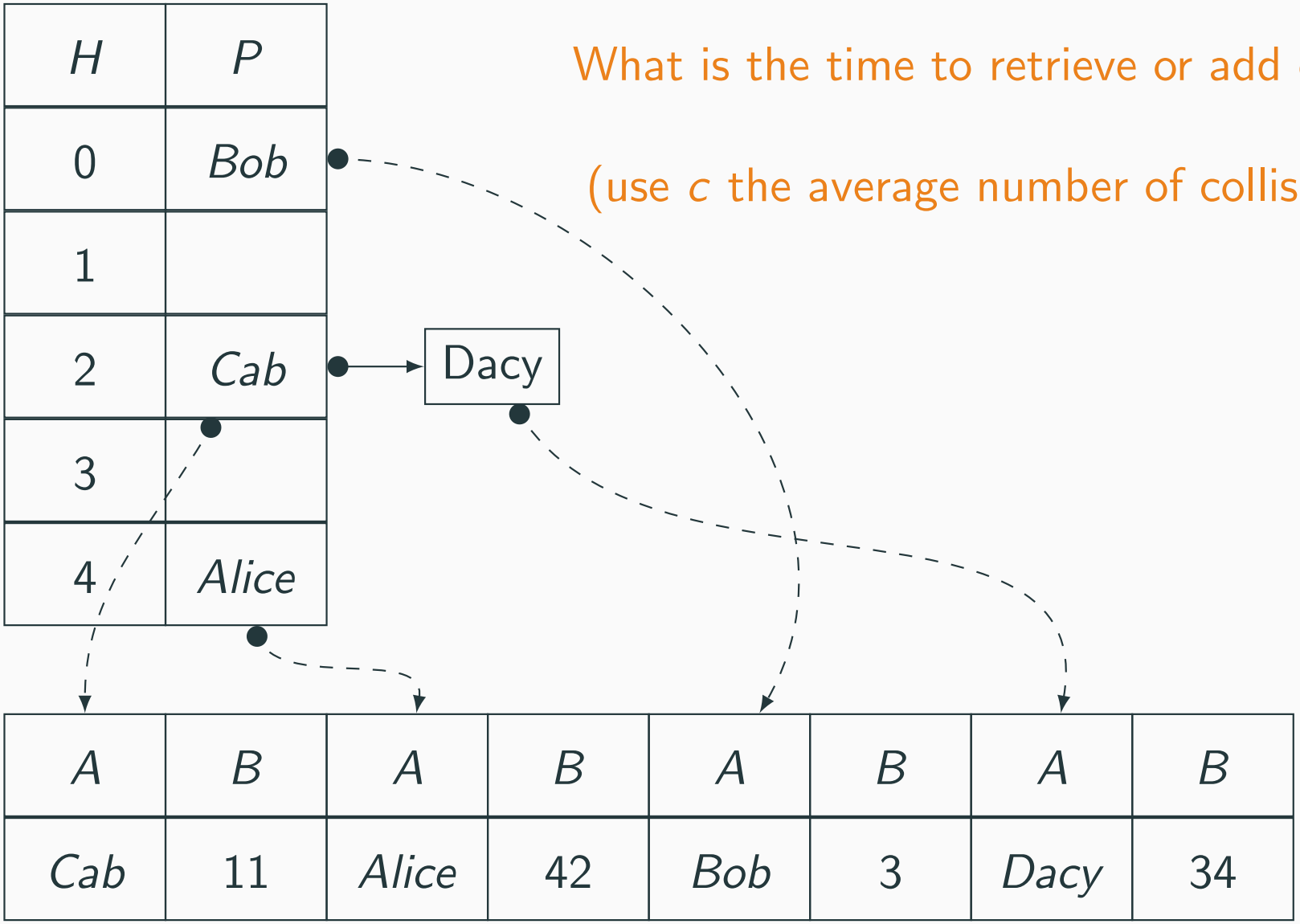
Add key x **to pointer** p

Append (x, p) to the list A(hash(x)).

Hash indexes



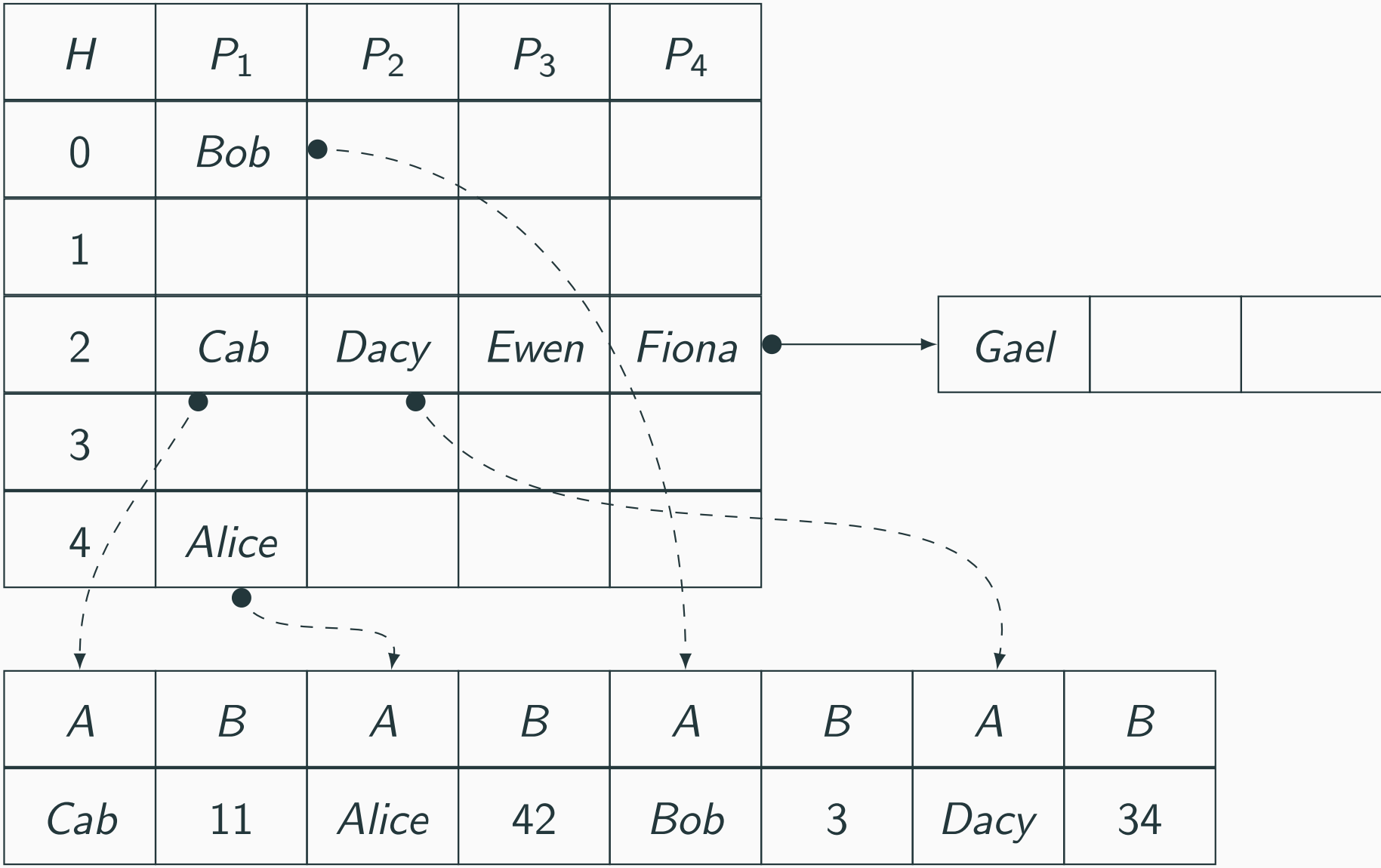
Hash indexes



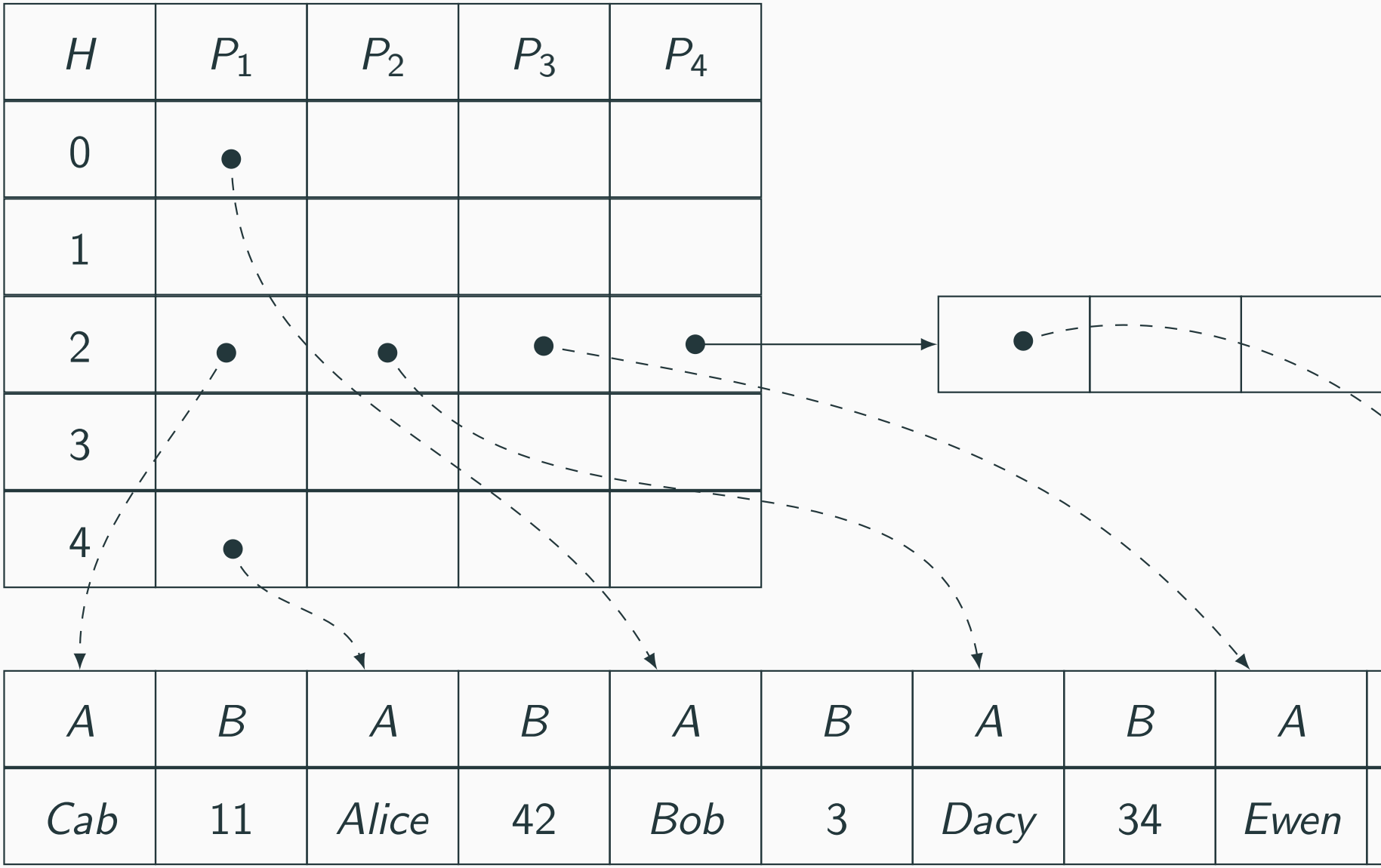
What is the time to retrieve or add data?

(use c the average number of collisions)

Dense hash indexes



Denser hash indexes



Pros & Cons of the denser hash indexes

Pros

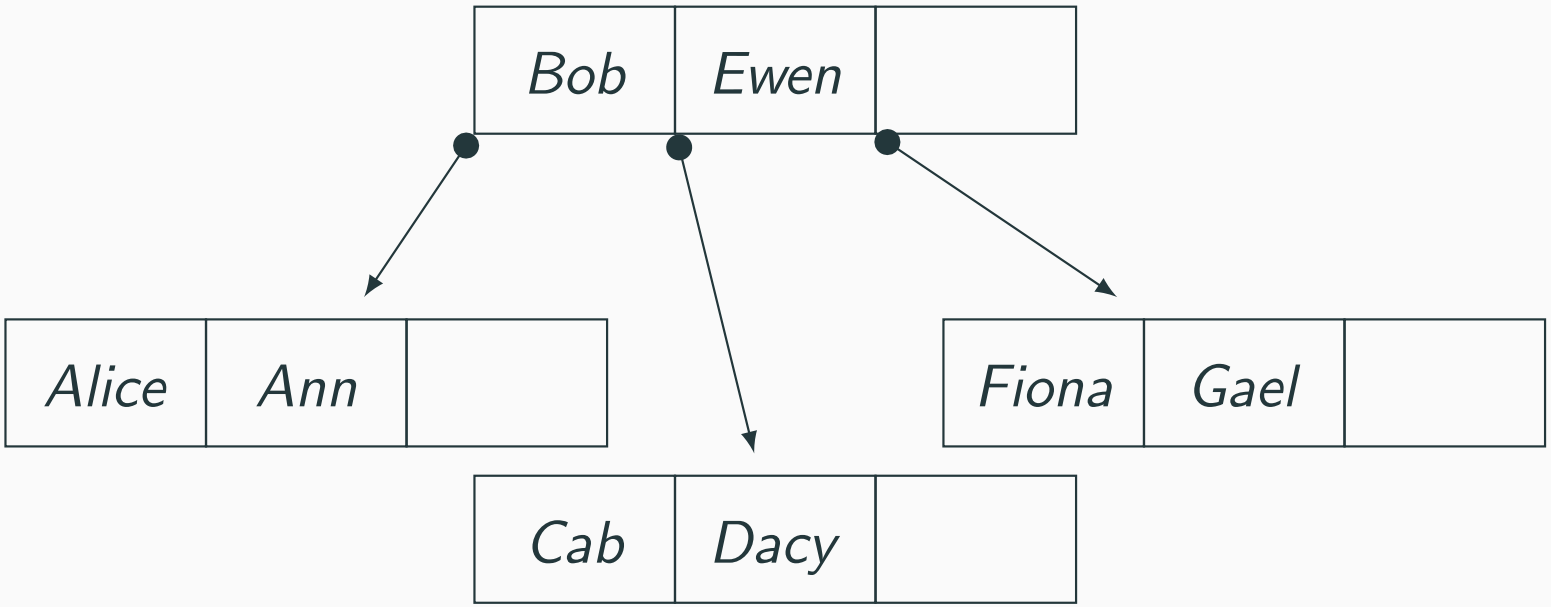
- Generally the fastest index (but with a small margin)
- Especially good on large datasets with complex types like strings
- Small space overhead

Cons

- At least two seeks required
- Cannot retrieve data in sorted order
- Can only check equality

B-trees

Just like binary search trees but B-ary trees.



Binary Search Trees (BST)

```
def search(key, node):
    if node is None or key == node.key:
        return node
    if key < node.key:
        return search(key, node.left)
    return search(key, node.right)
```

Insertion

Insertion code is similar to search but where we make sure that the tree stays balanced.

Deletion

Deletion is a bit tricky...

B-trees

Data

A search tree where all nodes (except the root) have an arity between $B/2$ and B.

Check existence of key x or retrieve tuples with $x \in [a; b]$

Recursively go down the tree. If we neglect CPU time this is $log_B(N)$ seeks. For large B we can expect 3 or 4 seeks at most.

Insert key x

Same as looking up except when a node contains strictly more than B elements, it is split into two. The splitter is added to the parent which can recursively trigger a splitting up to the root.

Delete key x

Same as insertion except we merge with neighbors when the node is too empty.

Exercise on B-trees

Describe the state of an initially empty B-tree with $B = 4$ after the following insertions:

- 12
- 52
- 24
- 67
- 89
- 55
- 30
- 20
- 5

Exercise on B-trees

Describe the state of an initially empty B-tree with $B = 4$ after the following insertions:

- 12
- 52
- 24
- 67
- 89
- 55
- 30
- 20
- 5

Exercise on B-trees

Describe the state of an initially empty B-tree with $B = 4$ after the following insertions:

- 12
- 52
- 24
- 67
- 89
- 55
- 30
- 20
- 5

Exercise on B-trees

Describe the state of an initially empty B-tree with $B = 4$ after the following insertions:

- 12
- 52
- 24
- 67
- 89
- 55
- 30
- 20
- 5

Exercise on B-trees

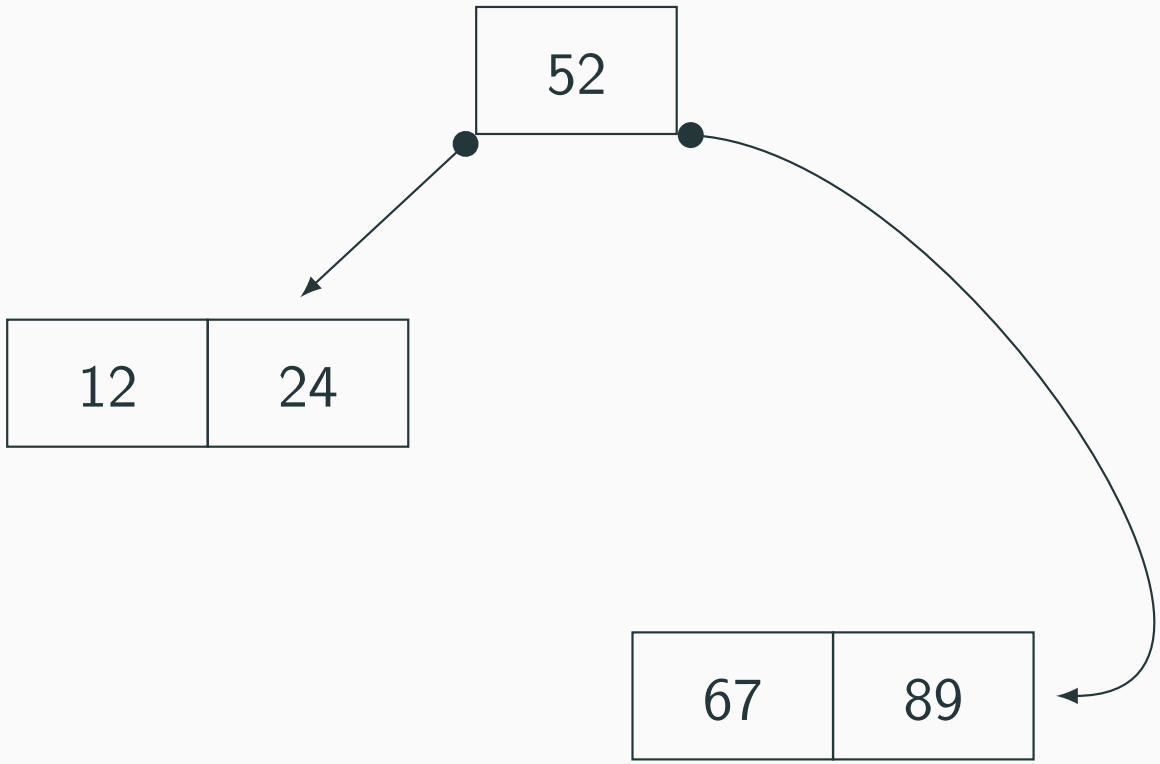
Describe the state of an initially empty B-tree with $B = 4$ after the following insertions:

- 12
- 52
- 24
- 67
- 89
- 55
- 30
- 20
- 5

Exercise on B-trees

Describe the state of an initially empty B-tree with $B = 4$ after the following insertions:

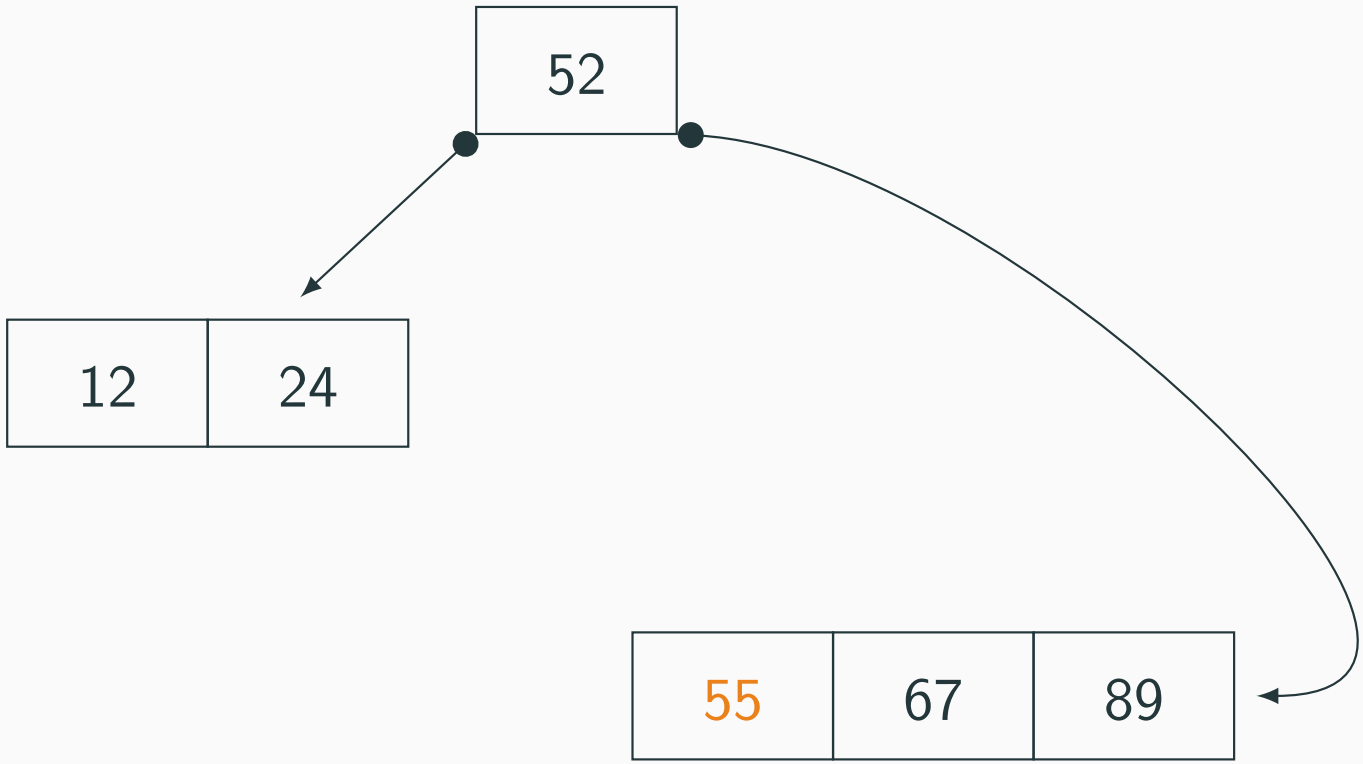
- 12
- 52
- 24
- 67
- 89
- 55
- 30
- 20
- 5



Exercise on B-trees

Describe the state of an initially empty B-tree with $B = 4$ after the following insertions:

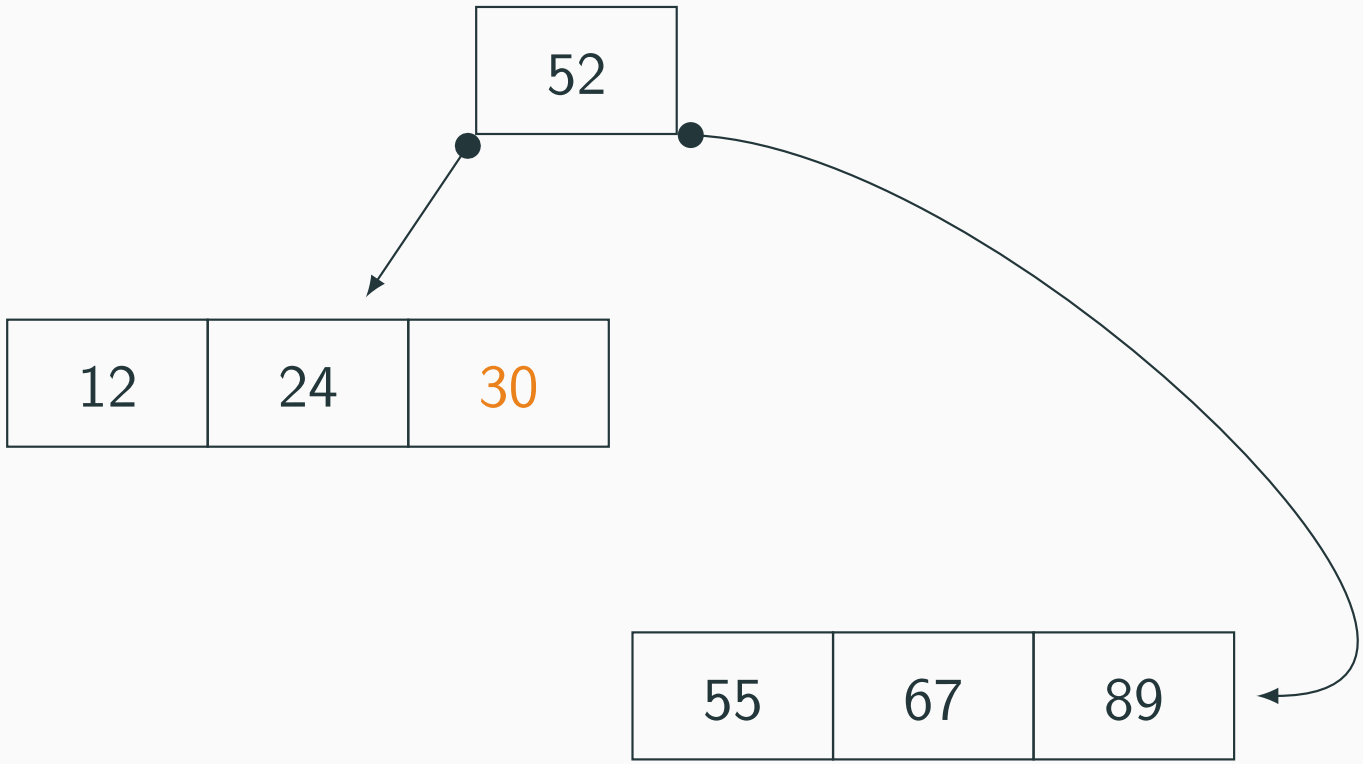
- 12
- 52
- 24
- 67
- 89
- 55
- 30
- 20
- 5



Exercise on B-trees

Describe the state of an initially empty B-tree with $B = 4$ after the following insertions:

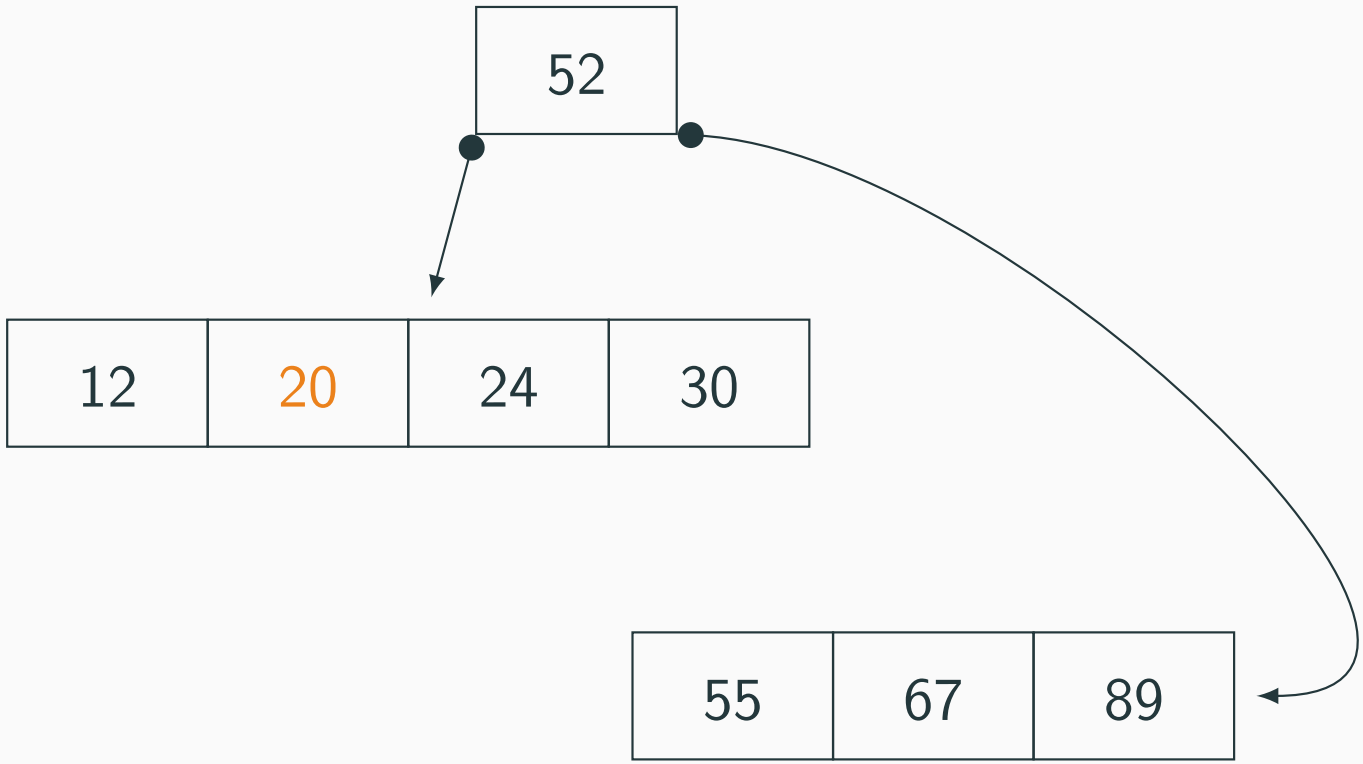
- 12
- 52
- 24
- 67
- 89
- 55
- 30
- 20
- 5



Exercise on B-trees

Describe the state of an initially empty B-tree with $B = 4$ after the following insertions:

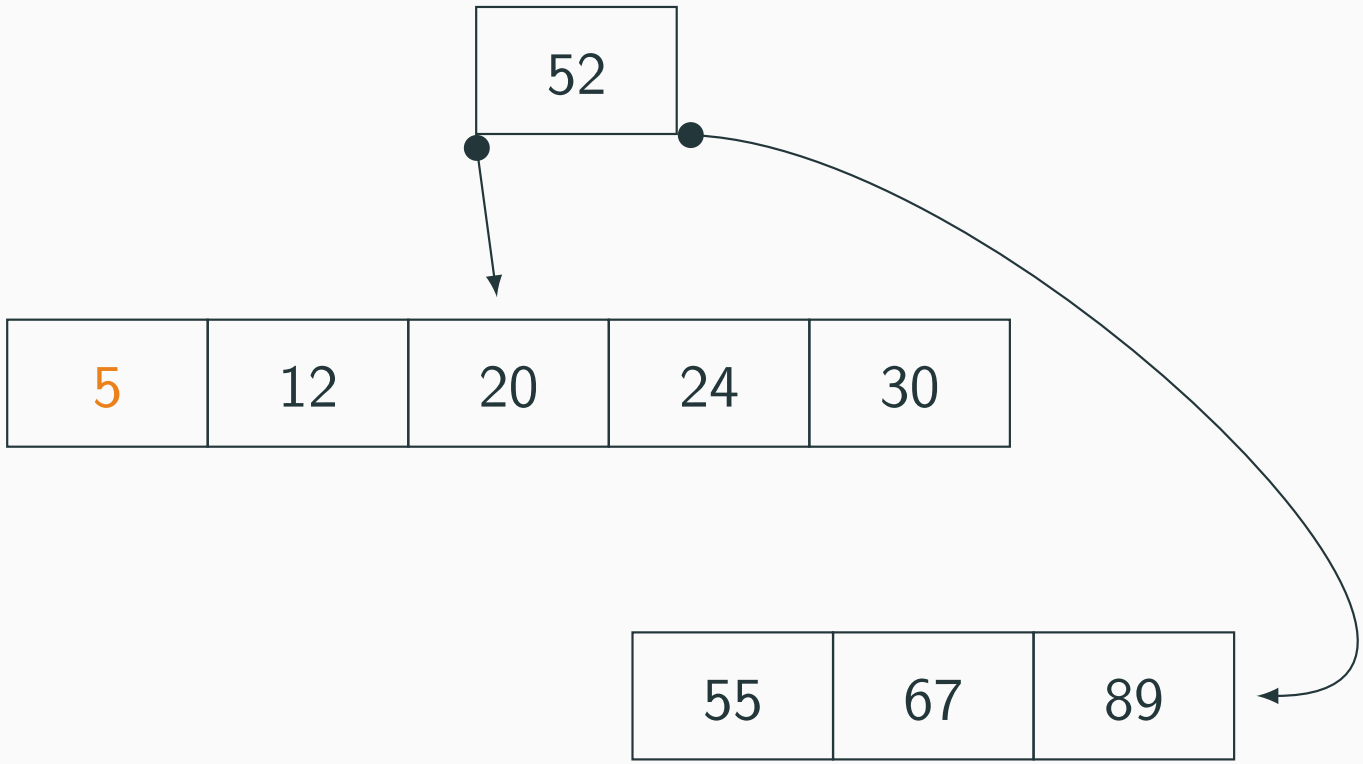
- 12
- 52
- 24
- 67
- 89
- 55
- 30
- 20
- 5



Exercise on B-trees

Describe the state of an initially empty B-tree with $B = 4$ after the following insertions:

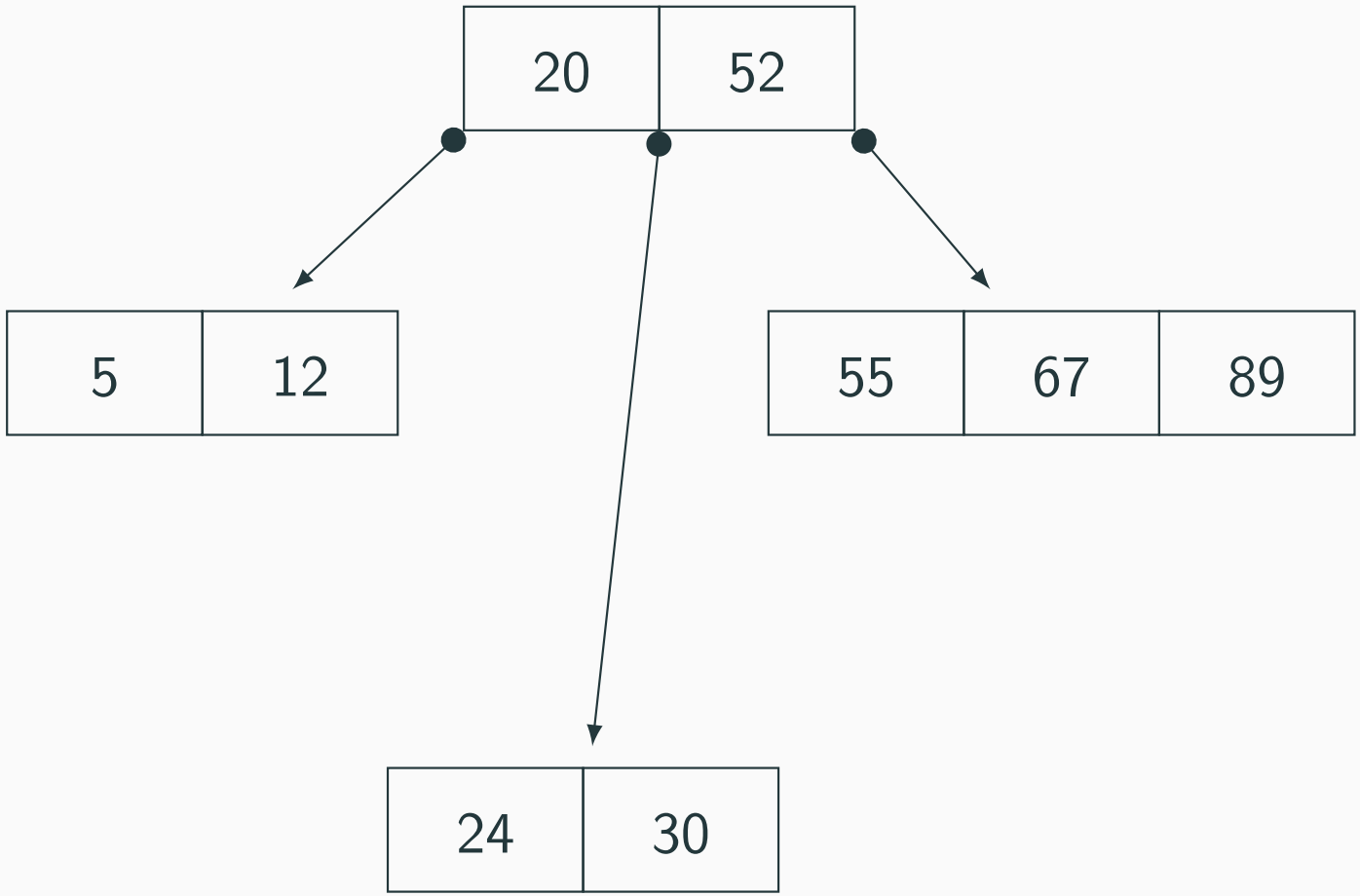
- 12
- 52
- 24
- 67
- 89
- 55
- 30
- 20
- 5



Exercise on B-trees

Describe the state of an initially empty B-tree with $B = 4$ after the following insertions:

- 12
- 52
- 24
- 67
- 89
- 55
- 30
- 20
- 5



Pros & Cons of the B-tree indexes

Pros

- Quite fast (especially on simple types)
- Retrieve data in order with very few seeks
- Can be used for prefixes / subkeys; a B-tree on (x, y) can be used to test whether a given x exists or retrieve the corresponding y.

Cons

- Larger number of seeks required
- Larger space overhead (especially for complex data types)

Generally the default index structure used!

Creating indexes

```
CREATE INDEX title_idx ON movies (name);

CREATE INDEX title_idx ON movies (lower(name)) USING hash;

CREATE UNIQUE INDEX ON movies (year,name);
```

Inverted Index

Inverted Index query

Given a set of sets $X_1, \ldots, X_n$ and an item i report the integers e such that $i \in X_e$.

You typically find them in cookbooks (recipes by ingredient)

Inverted Index

Inverted Index query

Given a set of sets $X_1, \ldots, X_n$ and an item i report the integers e such that $i \in X_e$.

You typically find them in cookbooks (recipes by ingredient)

Implementation

Typically a B-tree or a hash index giving for each i the set of e such that $i \in X_e$.

Inverted Index

Inverted Index query

Given a set of sets $X_1, \ldots, X_n$ and an item i report the integers e such that $i \in X_e$.

You typically find them in cookbooks (recipes by ingredient)

Implementation

Typically a B-tree or a hash index giving for each i the set of e such that $i \in X_e$.

Usage

They can be used for indexing JSON fields or getting records where some word appears in a string.

Usage of Generalized Inverted Indexes in PostgreSQL

```
CREATE TABLE movies (
      name VARCHAR(255), tags VARCHAR(16)[] );

INSERT INTO movies (name, tags)
   values ('Star Wars',
           '{"action", "adventure", "fantasy"}');

CREATE INDEX movie_tags ON movies USING gin (tags);

SELECT * FROM movies WHERE 'action' = ANY (tags);

SELECT * FROM movies
WHERE '{"action", "fantasy"}' && tags;
```

Usage of Generalized Inverted Indexes in PostgreSQL

```sql
CREATE INDEX lyrics_idx ON songs
    USING GIN (to_tsvector('english',lyrics));

SELECT name FROM songs
WHERE to_tsvector('english',lyrics) @@ to_tsquery('make');

--           name
-- --------------------------
-- Never Gonna Give You Up
-- It's Friday!
```

Bitmap indexes

Principle

For each value v store a bit array of length n (the number of records) where the i-th bit is set when the record has value v

Typical use case

- (Rarely) as an index built for values of low cardinality (gender, binary values, enum)
- (Very often) built for intermediate results of complex conditions (e.g. $x < 10$ or $y > 42$ and $z < 1$).

Bitmap indexes

Principle

For each value v store a bit array of length n (the number of records) where the i-th bit is set when the record has value v

Typical use case

- (Rarely) as an index built for values of low cardinality (gender, binary values, enum)
- (Very often) built for intermediate results of complex conditions (e.g. $x < 10$ or $y > 42$ and $z < 1$).

Can be built for pages instead of individual tuples.

Generalized Search Tree based indexes (GiST)

Principle

A GiST is a tree shaped index but where you cannot "split" efficiently into two non overlapping subsets.

Generalized Search Tree based indexes (GiST)

Principle

A GiST is a tree shaped index but where you cannot "split" efficiently into two non overlapping subsets.

Examples

Shapes in 2D, date intervals, text, etc.

Generalized Search Tree based indexes (GiST)

Principle

A GiST is a tree shaped index but where you cannot "split" efficiently into two non overlapping subsets.

Examples

Shapes in 2D, date intervals, text, etc.

Key idea

Split into two (or more) subsets with an overlapping.

Generalized Search Tree based indexes (GiST)

Principle

A GiST is a tree shaped index but where you cannot "split" efficiently into two non overlapping subsets.

Examples

Shapes in 2D, date intervals, text, etc.

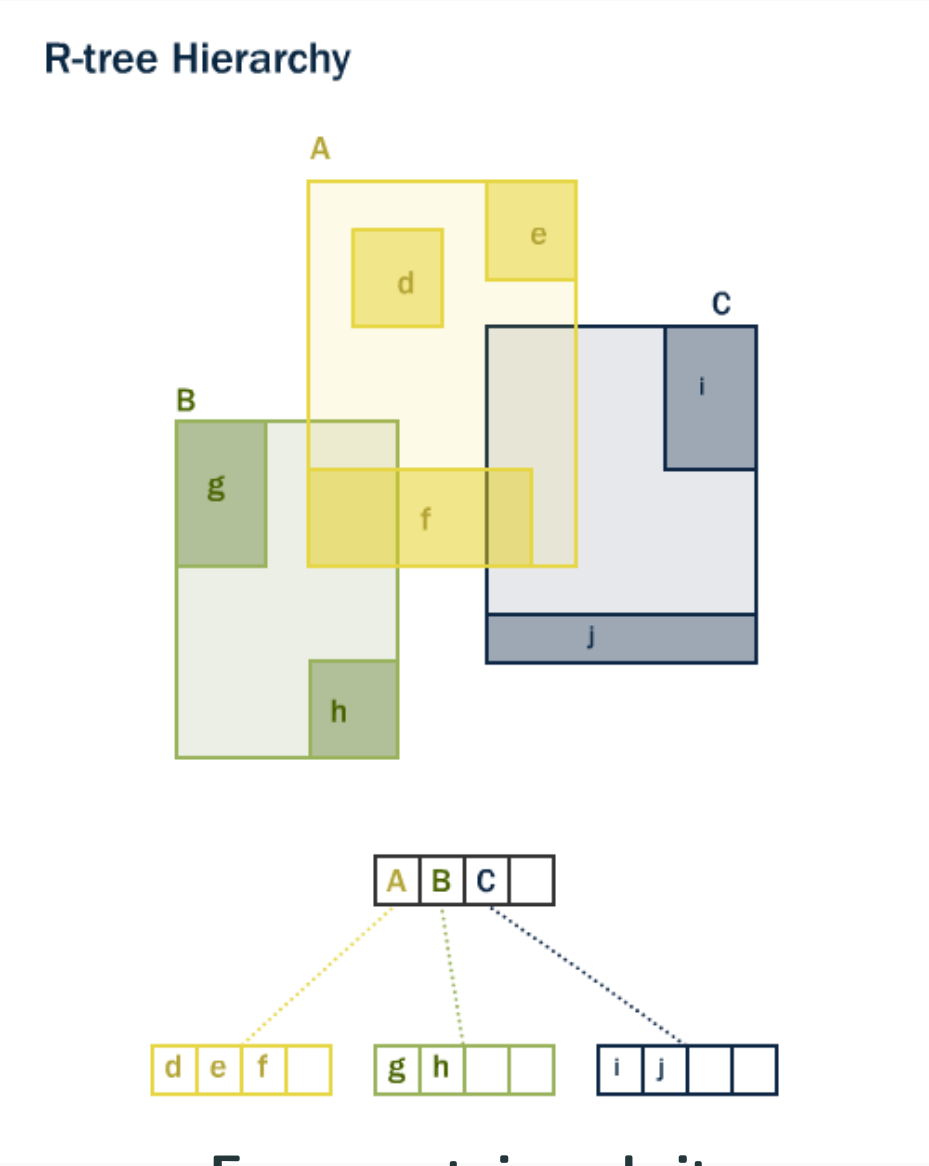
Key idea

Split into two (or more) subsets with an overlapping.

Consequence

Any search will go down on several parts of the tree, hence it is not as efficient as a B-tree!

Examples of GiST indexing



From postgis website

In conclusion

Queries containing patterns like $\texttt{FILTER}(\mathcal{R}, x = v)$ can often be drastically improved.

It is also used to speed up key constraints.

Computing joins

Joins

```sql
SELECT * FROM tableA, tableB WHERE idA=idB ;
```

```sql
SELECT * FROM tableA
WHERE EXISTS (
    SELECT 1 FROM  tableB
    WHERE idA=idB) ;
```

```sql
SELECT * FROM tableA LEFT JOIN tableB ON idA=idB ;
```

Computing joins

Joins can be computed with many different algorithms

- Nested loop join
- Block nested loop joins
- Index-join
- Sort-merge join
- Hash join
- Indexed-join
- and many others...

Nested loop join

The simplest join

```
for t1 in tableA:
  for t2 in tableB:
    if condition(t1,t2):
      output(t1,t2)
```

Nested loop join

The simplest join

```
for t1 in tableA:
  for t2 in tableB:
    if condition(t1,t2):
      output(t1,t2)
```

tableB will be read $|$tableA$|$ times...

Block nested loop join

A twist on the nested loop join:

```
for b1 in tableA.blocks
  for b2 in tableB.blocks
    for t1 in b1:
      for t2 in b2:
        if condition(t1,t2):
          output(t1,t2)
```

Same number of operations but each block is loaded once.

Index-join

Key idea

Imagine we have an index on `tB(id)` and we have the following query:

```
SELECT * FROM tA, tB WHERE tA.id=tB.id
```

We can use the index on `tB` to retrieve the matching tuples from `tA`.

Index-join

Key idea

Imagine we have an index on `tB(id)` and we have the following query:

```
SELECT * FROM tA, tB WHERE tA.id=tB.id
```

We can use the index on `tB` to retrieve the matching tuples from `tA`.

```
for t1 in tA
  for t2 in tB.idIndex(t1.id)
      output(t1,t2)
```

Index-join

What if the join condition is `tA.id = tB.id AND tA.x>tB.x`**?**

Index-join

What if the join condition is `tA.id = tB.id AND tA.x>tB.x`?

We can use the index on a weakened condition:

```
for t1 in tA
  for t2 in tB.idIndex(t1.id)
    if t1.x > t2.x:
       output(t1,t2)
```

Index-join

What if the join condition is `tA.id = tB.id AND tA.x>tB.x`?

We can use the index on a weakened condition:

```
for t1 in tA
  for t2 in tB.idIndex(t1.id)
    if t1.x > t2.x:
       output(t1,t2)
```

Or more generally:

```
for t1 in tableA
  for t2 in tableB.index[t1.value]:
     if condition(t1,t2):
        output(t1,t2)
```

Sort-merge-join

Another algorithm for the intersection of two lists

```
lA.sort()
lB.sort()
while len(lA)>0 and len(lB)>0:
  if lA[-1] = lB[-1]:
    output(lA[-1],lB[-1])
  if lA[-1] >= lB[-1]:
    lA.pop()
  else:
    lB.pop()
```

Sort-merge-join

More generally, for a sort-merge-join

- Get both inputs in sorted order
- Advance in both inputs by increasing order
- Output matching couples

Handles more than equality

Comparisons, max, min, ...

Hash-join

How to find the intersection of two sets A and B?

```
d = hashset()
res = []
for a in A:
  d.add(a)
for b in B:
  if b in d:
    res.append(b)
```

Hash-join

More generally, for a join with a condition C containing a set of equalities $x_1 = y_1 \wedge \cdots \wedge x_k = y_k$:

- For each tuple t_1 in `tableA` store it in a hash table with key $(t_1.x_1, \ldots, t_1.x_k)$,
- For each tuple t_2 in `tableB` look up the tuples t' in the hash table with key $(t_2.y_1, \ldots, t_2.y_k)$,
- If $C(t_1, t_2)$ output (t_1, t_2).

Indexed-join

Simple idea

Pre-compute and index the join result.

Indexed-join

Simple idea

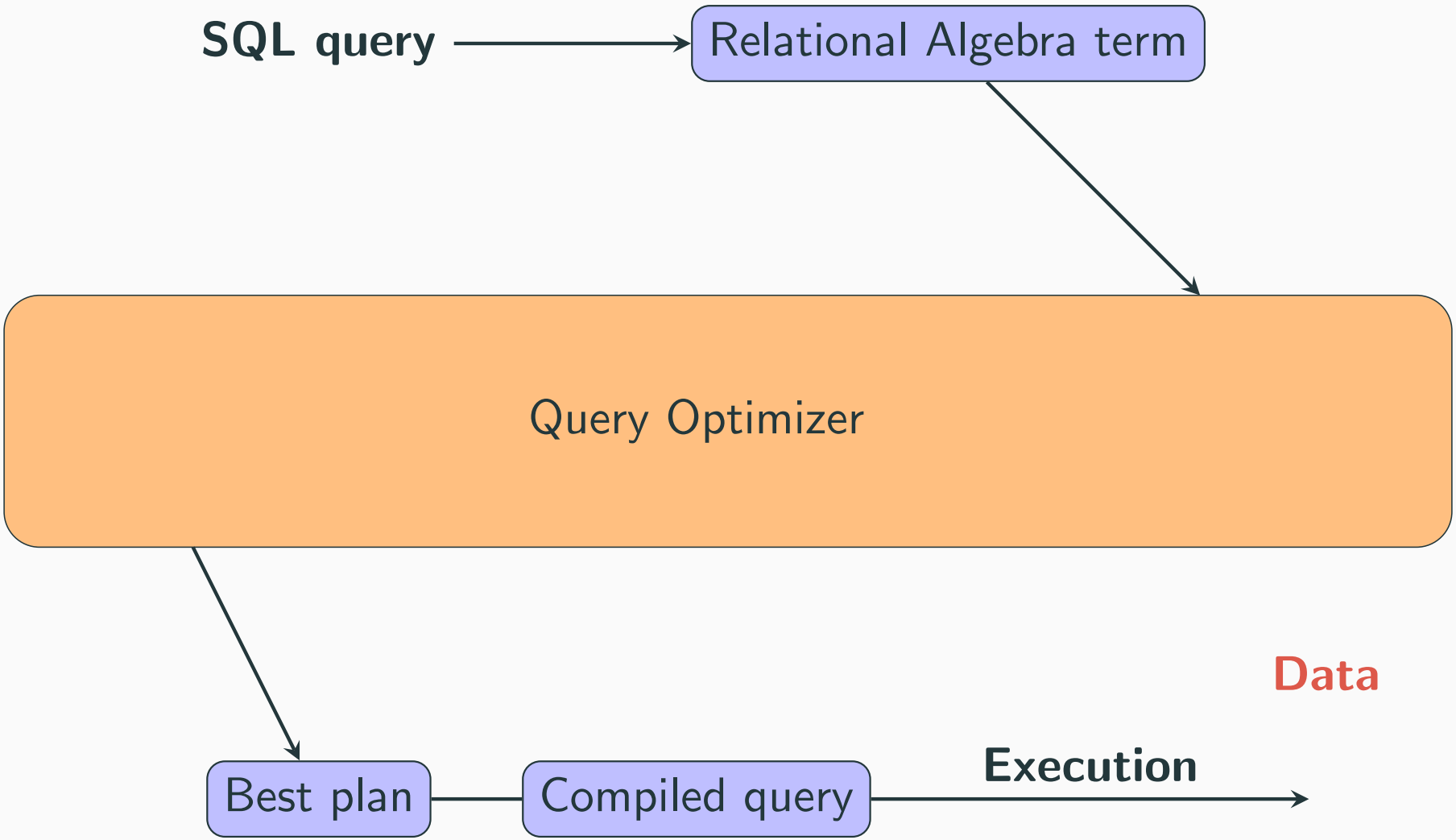
Pre-compute and index the join result.

Allows for very fast join but not without drawbacks

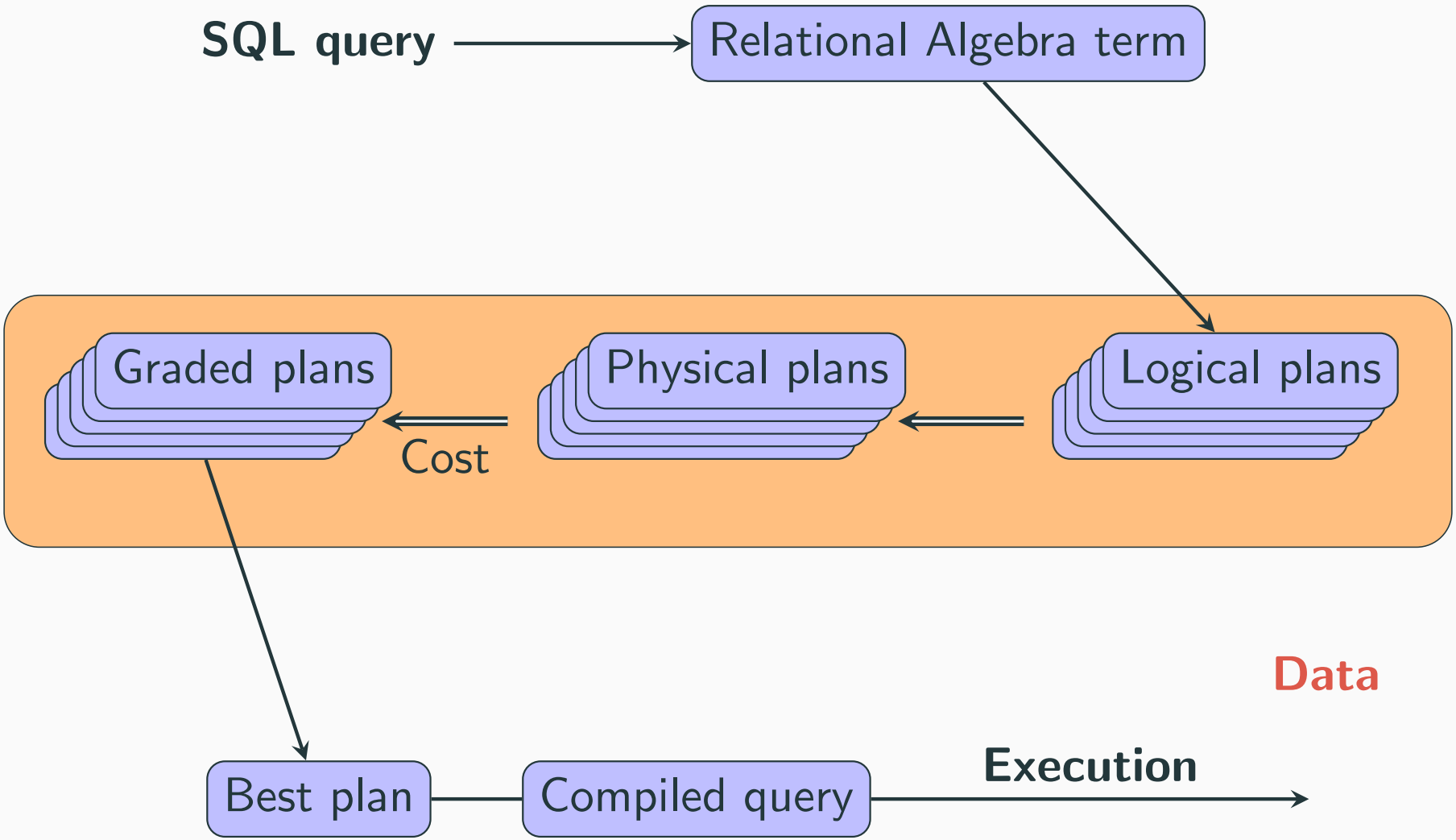
- Not available in most databases
- Slow to update
- Can eat up a lot of space

Only useful for very frequent joins with very few updates on the joined tables.

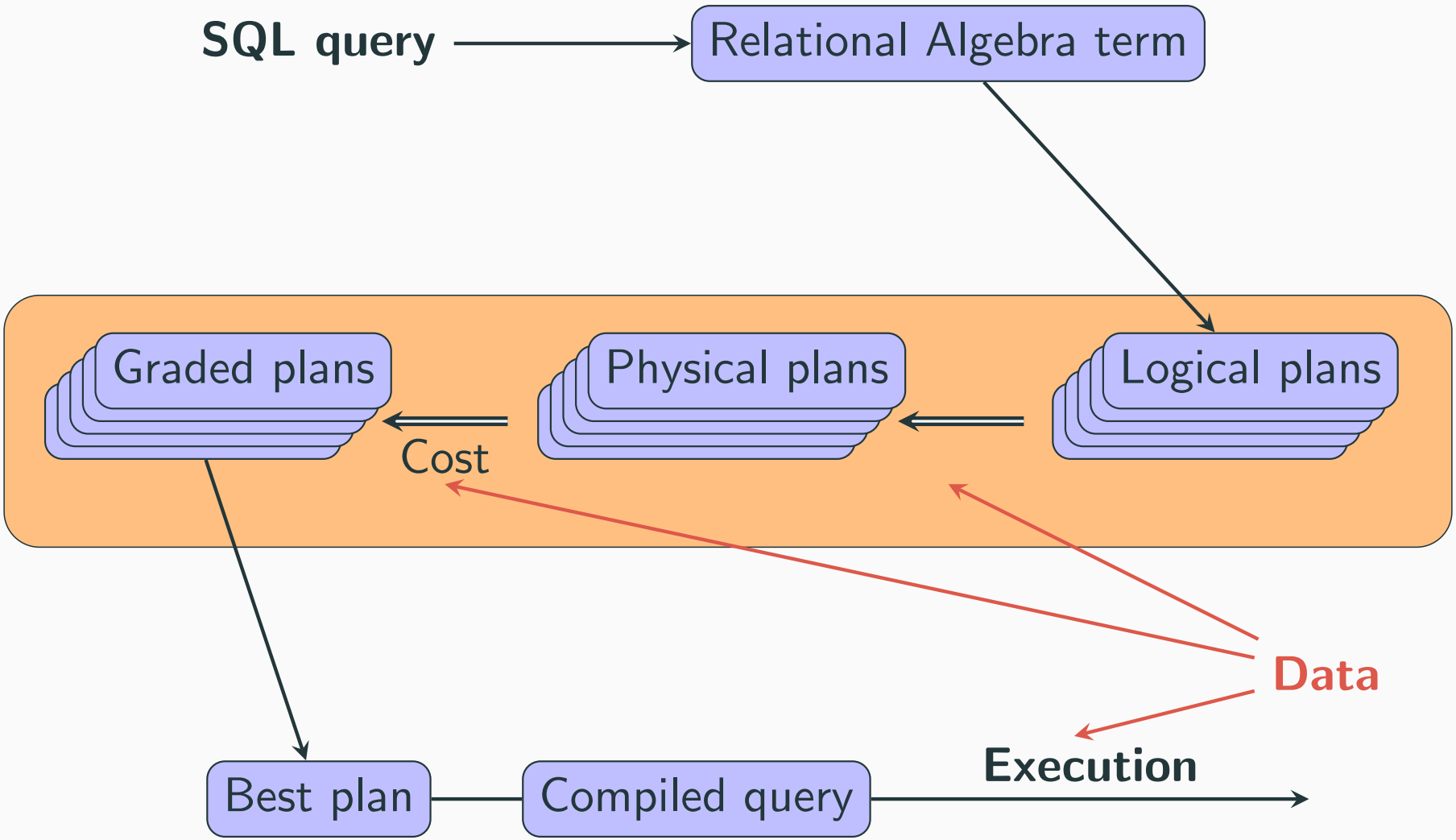
Query evaluation in a nutshell



Query evaluation in a nutshell



Query evaluation in a nutshell



Plans in Postgres

EXPLAIN

Most databases allow the user to see the query plan used

Generally with the `EXPLAIN COMMAND`.

Most databases allow the user to see the query plan used

Generally with the `EXPLAIN COMMAND`.

```
jachiet=> EXPLAIN SELECT name FROM songs WHERE to_tsvector('english',lyrics) @@ to_tsquery('make');
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on songs  (cost=8.25..12.76 rows=1 width=32)
   Recheck Cond: (to_tsvector('english'::regconfig, lyrics) @@ to_tsquery('make'::text))
    ->  Bitmap Index Scan on lyrics_idx  (cost=0.00..8.25 rows=1 width=0)
       Index Cond: (to_tsvector('english'::regconfig, lyrics) @@ to_tsquery('make'::text))
```

Most databases allow the user to see the query plan used

Generally with the `EXPLAIN COMMAND`.

```
jachiet=> EXPLAIN SELECT name FROM songs WHERE to_tsvector('english',lyrics) @@ to_tsquery('make');
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on songs  (cost=8.25..12.76 rows=1 width=32)
   Recheck Cond: (to_tsvector('english'::regconfig, lyrics) @@ to_tsquery('make'::text))
    ->  Bitmap Index Scan on lyrics_idx  (cost=0.00..8.25 rows=1 width=0)
          Index Cond: (to_tsvector('english'::regconfig, lyrics) @@ to_tsquery('make'::text))
```

Interesting to understand why some queries are slow:

- maybe the query is more complex than you thought?
- maybe your are missing some indexes?
- maybe Postgres selects a dumb query plan?

Adding ANALYZE makes the engine run the query and gather statistics

```
=> EXPLAIN ANALYZE SELECT name FROM songs WHERE to_tsvector('english',lyrics) @@ to_tsquery('make');
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on songs  (cost=0.00..2.02 rows=1 width=32) (actual time=1.412..3.126 rows=2 loops=1)
   Filter: (to_tsvector('english'::regconfig, lyrics) @@ to_tsquery('make'::text))
 Planning Time: 0.175 ms
 Execution Time: 3.154 ms
```

Effect of the cardinality estimation

```
jachiet=> EXPLAIN SELECT * FROM students WHERE age = 20 ;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on students  (cost=39.89..107.62 rows=1498 width=13)
   Recheck Cond: (age = 20)
   ->  Bitmap Index Scan on students_age_idx  (cost=0.00..39.52 rows=1498 width=0)
         Index Cond: (age = 20)
(4 rows)

jachiet=> EXPLAIN SELECT * FROM students WHERE age = 33 ;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using students_age_idx on students  (cost=0.29..8.22 rows=1 width=13)
   Index Cond: (age = 33)
(2 rows)
```

The various Postgres operators

- `SeqScan` explore the full table
- `Index Scan` explore the relevant part of a table using an index
- `Index Only Scan` explore the index
- `Bitmap Heap Scan` like a `SeqScan` but might skip some of the disk pages using a bitmap index
- `Bitmap Index Scan` builds a bitmap with an index
- `Bitmap Or` / `Bitmap And` builds a bitmap as the Or/And of two bitmaps
- `Filter`, `Nested loop`, `Hash join`, `Merge join`, self explanatory.
- `Materialize` we need to talk about the execution...

Execution of query plans

Database engines prefer to "stream" tuples rather than computing everything. For instance, for a filter:

Do

```
for t in subExpression:
  if cond(t):
    yield t
```

Don't

```
l = compute(subExpression)
return [e for e in l if cond(e)]
```

It avoids the materialization of large intermediate datasets

But it is sometimes unavoidable...

Conclusion

- Queries in databases are automatically optimized
- It works well up