CMPUT391

Overview of Query Execution and Optimization

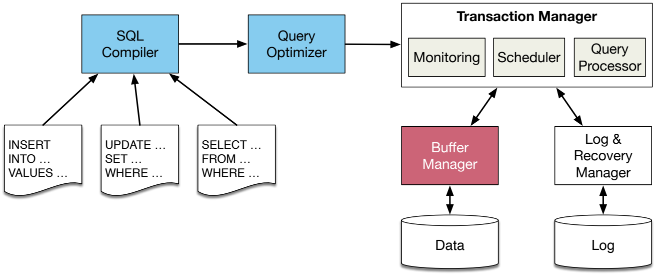
Instructor: Denilson Barbosa

University of Alberta

October 10, 2023

Slides by D. Barbosa, with suggested corrections and improvements by (in alphabetical order) C. Bins, D. Caminhas, K. Guhzva, Q. Lautischer, E. Macdonald, M. A. Nascimento, K. Newbury, M. Strobl, D. Sunderman, K. Wang, and K. Wong.

Simplified DBMS architecture



These notes look into algorithms and data structures used to answer relational queries.

SQL is a high level declarative language:

- A query specifies **what** the programmer wants...
- ... but it does not specify **how** to compute the answer.

We need to translate each SQL query into an an actual executable representation which the DBMS can process.

- Similar to generating executable bytecode from Java.

A **query plan** is an executable representation of a query.

- Query plans, essentially, relational algebra expressions.
- Recall that we can always translate SQL queries into the algebra because they are equivalent[1].

[1] Actual implementations of the algebra have operators for advanced SQL features like recursion.

For example, the SQL query

```
SELECT m.director FROM Movie m, Cast c
WHERE m.title=c.title AND m.year=c.year AND
      c.actor='Bill Murray'
```

is equivalent to $\pi_{\texttt{director}}\left(\sigma_{\texttt{actor='Bill Murray'}}\left(\texttt{Movie} \bowtie \texttt{Cast}\right)\right)$

Note that we can represent every algebra expressions as a tree.

It is common to represent query plans as trees. Most DBMSs have GUIs to let the DBA inspect plans in this way.

$$\begin{tikzcd}
& \pi_{\texttt{director}} & \\
& \sigma_{\texttt{actor='Bill Murray'}} \arrow[u] & \\
& \bowtie \arrow[u] & \\
\texttt{Movie} \arrow[ur] & & \texttt{Cast} \arrow[ul]
\end{tikzcd}$$

The difference between a plan and an algebra expression is that the plan specifies **access methods** are used to read the data and which **algorithm** is used to implement each operation[2].

$$\begin{tikzcd}
& \pi_{\texttt{director}} & \\
& \sigma_{\texttt{actor='Bill Murray'}} \arrow[u] & \\
& \bowtie \arrow[u] & \\
\texttt{Movie} \arrow[ur] & & \texttt{Cast} \arrow[ul]
\end{tikzcd}
\qquad
\begin{tikzcd}
& \pi_{\texttt{director}} & \\
& \sigma_{\texttt{actor='Bill Murray'}} \arrow[u] & \\
& \texttt{nested\_loop\_join} \arrow[u] & \\
\texttt{scan(Movie)} \arrow[ur] & & \texttt{scan(Cast)} \arrow[ul]
\end{tikzcd}$$

The `scan()` **operator** is a full table scan, which pulls tuples from a table file (heap or sequential).

[2]We will see that that are often many choices, especially for joins.

The life cycle of a query

```c
#include <stdio.h>
#include <sqlite3.h>

int main(int argc, char **argv){
    sqlite3 *db; //the database
    sqlite3_stmt *stmt; //the SQL statement
    int rc; //the return code

    rc = sqlite3_open(argv[1], &db);
    if( rc ){
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db); return 1;
    }

    char *sql_stmt = "SELECT title,year FROM Movie WHERE director='Kubrik'";
    rc = sqlite3_prepare_v2(db, sql_stmt, -1, &stmt, 0);

    if (rc != SQLITE_OK) {
        fprintf(stderr, "Preparation failed: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db); return 1;
    }

    while((rc = sqlite3_step(stmt)) == SQLITE_ROW) {
        int col;
        for(col=0; col<sqlite3_column_count(stmt)-1; col++) {
            printf("%s|", sqlite3_column_text(stmt, col));
        }
        printf("%s", sqlite3_column_text(stmt, col));
        printf("\n");
    }
    sqlite3_finalize(stmt);
}
```

The life cycle of a query

```c
#include <stdio.h>
#include <sqlite3.h>

int main(int argc, char **argv){
    sqlite3 *db; //the database
    sqlite3_stmt *stmt; //the SQL statement
    int rc; //the return code

    rc = sqlite3_open(argv[1], &db);
    if( rc ){
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db); return 1;
    }

    char *sql_stmt = "SELECT title,year FROM Movie WHERE director='Kubrik'";
    rc = sqlite3_prepare_v2(db, sql_stmt, -1, &stmt, 0);

    if (rc != SQLITE_OK) {
        fprintf(stderr, "Preparation failed: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db); return 1;
    }

    while((rc = sqlite3_step(stmt)) == SQLITE_ROW) {
        int col;
        for(col=0; col<sqlite3_column_count(stmt)-1; col++) {
            printf("%s|", sqlite3_column_text(stmt, col));
        }
        printf("%s", sqlite3_column_text(stmt, col));
        printf("\n");
    }
    sqlite3_finalize(stmt);
}
```

compile SQL query into (algebraic) plan

The life cycle of a query

```c
#include <stdio.h>
#include <sqlite3.h>

int main(int argc, char **argv){
    sqlite3 *db; //the database
    sqlite3_stmt *stmt; //the SQL statement
    int rc; //the return code

    rc = sqlite3_open(argv[1], &db);
    if( rc ){
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db); return 1;
    }

    char *sql_stmt = "SELECT title,year FROM Movie WHERE director='Kubrik'";
    rc = sqlite3_prepare_v2(db, sql_stmt, -1, &stmt, 0);

    if (rc != SQLITE_OK) {
        fprintf(stderr, "Preparation failed: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db); return 1;
    }

    while((rc = sqlite3_step(stmt)) == SQLITE_ROW) {
        int col;
        for(col=0; col<sqlite3_column_count(stmt)-1; col++) {
            printf("%s|", sqlite3_column_text(stmt, col));
        }
        printf("%s", sqlite3_column_text(stmt, col));
        printf("\n");
    }
    sqlite3_finalize(stmt);
}
```

compile SQL query into (algebraic) plan

fetch tuples one-at-a-time

Compiling the SQL query

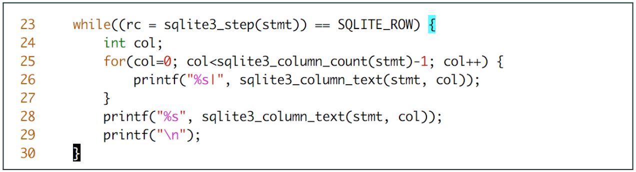
```c
    char *sql_stmt = "SELECT title,year FROM Movie WHERE director='Kubrik'";
    rc = sqlite3_prepare_v2(db, sql_stmt, -1, &stmt, 0);
```

… produces a query plan looking like this:

$$\begin{tikzcd}
\pi_{\texttt{title,year}} \\
\sigma_{\texttt{director='Kubrick'}} \arrow[u] \\
\texttt{scan(Movie)} \arrow[u]
\end{tikzcd}$$

The query plan is executed **iterately**:

Every call to `sqlite3_step()` fetches **one new tuple** of the answer, and stores it in the heap space of the application.



```c
while((rc = sqlite3_step(stmt)) == SQLITE_ROW) {
    int col;
    for(col=0; col<sqlite3_column_count(stmt)-1; col++) {
        printf("%s|", sqlite3_column_text(stmt, col));
    }
    printf("%s", sqlite3_column_text(stmt, col));
    printf("\n");
}
```

`sqlite3_step()` has a specific return code to indicate that all tuples in the answer have already been fetched.

How is a call to `sqlite3_step()` executed?

Every node in the query plan implements an interface defining the `getNext()` method, which must returns a tuple that has not been returned **by that node** before.

$$\begin{tikzcd}
\pi_{\texttt{title,year}} \\
\sigma_{\texttt{director='Kubrick'}} \arrow[u] \\
\texttt{scan(Movie)} \arrow[u]
\end{tikzcd}$$

`sqlite3_step()` calls `getNext()` on the root node of the plan, which leads to further calls to the nodes below, until a new tuple is returned to the application.

How is a call to `sqlite3_step()` executed?

Every node in the query plan implements an interface defining the `getNext()` method, which must returns a tuple that has not been returned **by that node** before.

$$\begin{tikzcd}
\pi_{\texttt{title,year}} \\
\sigma_{\texttt{director='Kubrick'}} \arrow[u] \\
\texttt{scan(Movie)} \arrow[u]
\end{tikzcd}
\qquad
\begin{tikzcd}
\texttt{sqlite3\_step()} \arrow[d] \\
\texttt{getNext()}
\end{tikzcd}$$

`sqlite3_step()` calls `getNext()` on the root node of the plan, which leads to further calls to the nodes below, until a new tuple is returned to the application.

How is a call to `sqlite3_step()` executed?

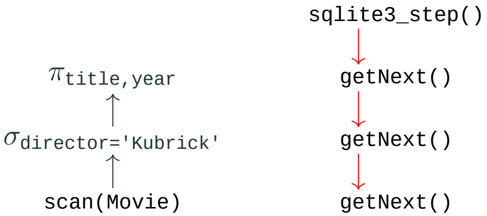
Every node in the query plan implements an interface defining the `getNext()` method, which must returns a tuple that has not been returned **by that node** before.

$$\begin{tikzcd}
 & \texttt{sqlite3\_step()} \arrow[d] \\
\pi_{\texttt{title,year}} & \texttt{getNext()} \arrow[d] \\
\sigma_{\texttt{director='Kubrick'}} \arrow[u] & \texttt{getNext()} \\
\texttt{scan(Movie)} \arrow[u] &
\end{tikzcd}$$

`sqlite3_step()` calls `getNext()` on the root node of the plan, which leads to further calls to the nodes below, until a new tuple is returned to the application.

How is a call to `sqlite3_step()` executed?

Every node in the query plan implements an interface defining the `getNext()` method, which must returns a tuple that has not been returned **by that node** before.



$\pi_{\texttt{title,year}}$
$\sigma_{\texttt{director='Kubrick'}}$
scan(Movie)
sqlite3_step()
getNext()
getNext()
getNext()

`sqlite3_step()` calls `getNext()` on the root node of the plan, which leads to further calls to the nodes below, until a new tuple is returned to the application.

How is a call to `sqlite3_step()` executed?

Every node in the query plan implements an interface defining the `getNext()` method, which must returns a tuple that has not been returned **by that node** before.

$$\begin{tikzcd}
\pi_{\texttt{title,year}} \\
\sigma_{\texttt{director='Kubrick'}} \arrow[u] \\
\texttt{scan(Movie)} \arrow[u]
\end{tikzcd}$$

$$\begin{tikzcd}
\texttt{sqlite3\_step()} \arrow[d] \\
\texttt{getNext()} \arrow[d] \\
\texttt{getNext()} \arrow[d, shift right] \\
\texttt{getNext()} \arrow[u, shift right]
\end{tikzcd}$$

`sqlite3_step()` calls `getNext()` on the root node of the plan, which leads to further calls to the nodes below, until a new tuple is returned to the application.

How is a call to `sqlite3_step()` executed?

Every node in the query plan implements an interface defining the `getNext()` method, which must returns a tuple that has not been returned **by that node** before.

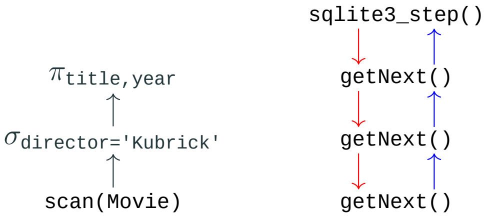
$$\begin{tikzcd}
\pi_{\texttt{title,year}} \\
\sigma_{\texttt{director='Kubrick'}} \arrow[u] \\
\texttt{scan(Movie)} \arrow[u]
\end{tikzcd}$$

$$\begin{tikzcd}
\texttt{sqlite3\_step()} \arrow[d] \\
\texttt{getNext()} \arrow[d, shift right] \\
\texttt{getNext()} \arrow[u, shift right] \arrow[d, shift right] \\
\texttt{getNext()} \arrow[u, shift right]
\end{tikzcd}$$

`sqlite3_step()` calls `getNext()` on the root node of the plan, which leads to further calls to the nodes below, until a new tuple is returned to the application.

How is a call to `sqlite3_step()` executed?

Every node in the query plan implements an interface defining the `getNext()` method, which must returns a tuple that has not been returned **by that node** before.



$\pi_{\text{title,year}}$
$\sigma_{\text{director='Kubrick'}}$
scan(Movie)
sqlite3_step()
getNext()
getNext()
getNext()

`sqlite3_step()` calls `getNext()` on the root node of the plan, which leads to further calls to the nodes below, until a new tuple is returned to the application.

Those objects returning the “next tuple” are called **iterators.** The DBMS implements at lest one iterator for each kind of operator in the query plan.

The DBMS might have different iterators with different algorithms for the same operator. At query optimization time, the DBMS must decide which iterator to use based on the cost of the corresponding algorithm and the available resources.

Iterators are **stateful** objects: they cannot return the same tuple more than once, so they need to somehow remember which ones were already returned.

- Iterators return **`<EOF>`** (end of file) when there are no more tuples to be returned.

Query Compilation

Compiling SQL into algebra expressions

Recap:

- SQL is a complex language, with many sophisticated constructs that are beyond our scope here. Instead we focus on the basic building blocks.
- A query plan is, essentially, an algebra expression with access methods instead of table names.
- The main difference between a plan and a “pure” algebra expression is that in the plan we need to specify the access methods we use to read the database.

Compiling basic SQL expressions

The basic compilation step translates a **conjunctive SQL query** into a “canonical” plan like:

SELECT $a_1, a_2, \ldots, a_n$
FROM T_1 [, T_2 [, $\ldots$ [, T_m]]]
WHERE c_1 **AND** c_2 **AND** … **AND** c_k

$$\begin{tikzcd}
 & & & & \pi_{a_1,\ldots,a_n} & \\
 & & & & \sigma_{c_1 \wedge \cdots \wedge c_k} \arrow[u] & \\
 & & & & \times \arrow[u] & \\
 & & & \times \arrow[ur] & & T_m \arrow[ul] \\
 & & \cdots \arrow[ur] & & T_{m-1} \arrow[ul] & \\
 & \times \arrow[ur] & & & & \\
T_1 \arrow[ur] & & T_2 \arrow[ul] & & &
\end{tikzcd}$$

If there is a single table expression in the query, we omit the Cartesian product of course.

We omit the `scan()` operators here for simplicity.

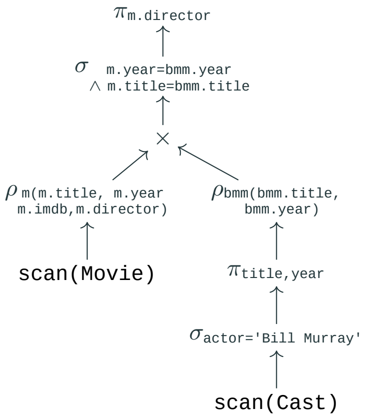
The *renaming operator* $\rho()$ is used to handle the tuple variable declarations in SQL.

```sql
SELECT m.title, m.year
FROM Movie m
WHERE m.director = 'Ivan Reitman'
```

$$\begin{tikzcd}
\pi_{\texttt{m.title, m.year}} \\
\sigma_{\texttt{m.director='Ivan Reitman'}} \arrow[u] \\
\rho_{\texttt{m}(\texttt{m.title, m.year, m.imdb, m.director})} \arrow[u] \\
\texttt{scan(Movie)} \arrow[u]
\end{tikzcd}$$

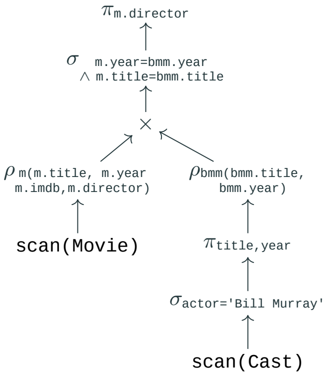
Table expressions given as SQL queries are translated into **sub-expressions** “below the Cartesian product” in the main plan:

```
SELECT m.director
FROM Movie m,
  (SELECT title, year
   FROM Cast
   WHERE actor='Bill Murray'
  ) AS bmm
WHERE m.year=bmm.year
  AND m.title=bmm.title;
```

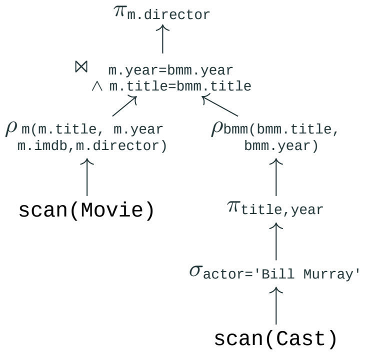


$\pi_{\texttt{m.director}}$
$\sigma_{\texttt{m.year=bmm.year} \wedge \texttt{m.title=bmm.title}}$
$\times$
$\rho_{\texttt{m(m.title, m.year m.imdb,m.director)}}$
$\rho_{\texttt{bmm(bmm.title, bmm.year)}}$
scan(Movie)
$\pi_{\texttt{title,year}}$
$\sigma_{\texttt{actor='Bill Murray'}}$
scan(Cast)

It is OK to "push down" conditions in the `WHERE` clause that define a join, replacing the Cartesian product when possible:



π m.director
σ m.year=bmm.year ∧ m.title=bmm.title
×
ρ m(m.title, m.year m.imdb,m.director)
ρ bmm(bmm.title, bmm.year)
scan(Movie)
π title,year
σ actor='Bill Murray'
scan(Cast)



π m.director
⋈ m.year=bmm.year ∧ m.title=bmm.title
ρ m(m.title, m.year m.imdb,m.director)
ρ bmm(bmm.title, bmm.year)
scan(Movie)
π title,year
σ actor='Bill Murray'
scan(Cast)

Set/bag operators are also translated into their algebra counterparts:

Example:

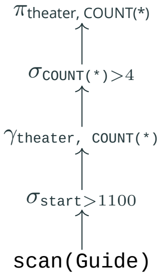
```sql
SELECT title, year
FROM Cast
WHERE actor="Bill Murray"
INTERSECT
SELECT title, year
FROM Movie
WHERE imdb>7
```

$$\begin{tikzcd}
& \cap_{\text{distinct}} & \\
\pi_{\texttt{title,year}} \arrow[ur] & & \pi_{\texttt{title,year}} \arrow[ul] \\
\sigma_{\texttt{actor='Bill Murray'}} \arrow[u] & & \sigma_{\texttt{imdb>7}} \arrow[u] \\
\texttt{scan(Cast)} \arrow[u] & & \texttt{scan(Movie)} \arrow[u]
\end{tikzcd}$$

Aggregation is handled by operator $\gamma_{\text{<grouping>,fn()}}(R)$

The **GROUP BY** clause goes into “grouping” and `fn()` is the set function. The **HAVING** clause are handled with a selection.

```
SELECT theater, COUNT(*)
FROM Guide
WHERE start > 1100
GROUP BY theater
HAVING COUNT(*) > 4
```


$$\begin{tikzcd}
\pi_{\text{theater, COUNT(*)}} \\
\sigma_{\texttt{COUNT(*)}>4} \arrow[u] \\
\gamma_{\texttt{theater, COUNT(*)}} \arrow[u] \\
\sigma_{\texttt{start}>1100} \arrow[u] \\
\texttt{scan(Guide)} \arrow[u]
\end{tikzcd}$$

Sub-Queries in Other Expressions

Recall SQL allows sub-queries to appear inside value expressions and in predicates in the `WHERE` clause.

```
SELECT 100 + (
        SELECT COUNT(*)
        FROM Movie
       );
```

```
SELECT m.title
FROM Movie m
WHERE m.imdb = (SELECT MAX(imdb)
                FROM Movie);
```

Actual query plans in production DBMSs have nodes beyond the relational algebra (e.g., to handle complex arithmetic or string operations from the results of sub-queries).

The SQL Preprocessor

Other constructs in SQL such as *views* and Common Table Expressions (CTEs) given through `WITH` clauses are handled by the SQL **preprocessor**.

Views and non-recursive CTEs are treated essentially as *macros* in a C program: the preprocessor replaces every occurrence of the CTE or the view in a table expression by that CTE/view definition:

```
WITH TopMovie(title, year) AS (
    SELECT title, year
    FROM Movie WHERE imdb > 7
)
SELECT c.role
FROM Cast c, TopMovie tm
WHERE c.title=tm.title AND
      c.year=tm.year AND
      c.actor="Sigourney Weaver";
```

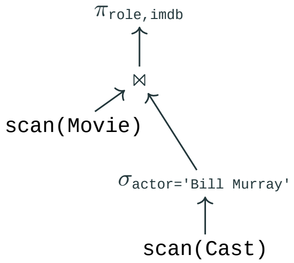
```
SELECT c.role
FROM Cast c, (
    SELECT title, year
    FROM Movie WHERE imdb > 7
    ) AS tm
WHERE c.title=tm.title AND
      c.year=tm.year AND
      c.actor="Sigourney Weaver";
```

The Iterator Model

The Iterator Interface

All nodes in a query plan must implement the following three methods:

- **Open()**: creates/requests the necessary data structures.
- **GetNext()**: produces **the** next tuple for that node.
- **Close()**: releases the necessary data structures.



Remember this

Each call to **GetNext()** produces just one tuple: the next tuple that the node must return.

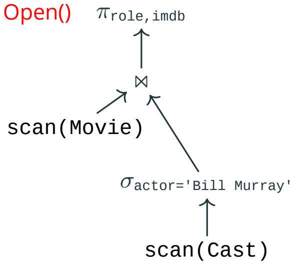
Open()/Close()

Open() is called recursively, top-down, from the root of the tree

A call to **Open()** is supposed to:

(1) create any necessary input buffers
(2) initialize any variables needed (ex: pointers to tuples in a nested loop join operator)
(3) open a file if the operator is a table or index scan

The analogous **Close()** method frees up all those resources.



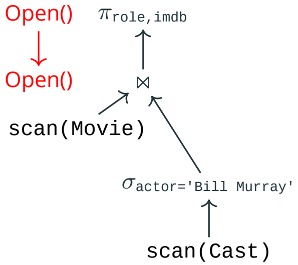
Open()/Close()

Open() is called recursively, top-down, from the root of the tree

A call to **Open()** is supposed to:

1. create any necessary input buffers
2. initialize any variables needed (ex: pointers to tuples in a nested loop join operator)
3. open a file if the operator is a table or index scan

The analogous **Close()** method frees up all those resources.



Open()/Close()

Open() is called recursively, top-down, from the root of the tree

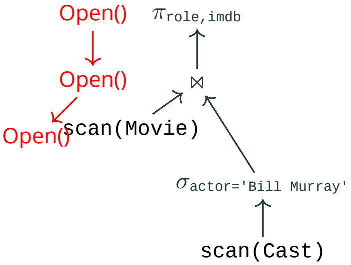
A call to **Open()** is supposed to:

(1) create any necessary input buffers

(2) initialize any variables needed (ex: pointers to tuples in a nested loop join operator)

(3) open a file if the operator is a table or index scan

The analogous **Close()** method frees up all those resources.


Open()
Open()
Open()
$\pi_{\texttt{role,imdb}}$
$\bowtie$
scan(Movie)
$\sigma_{\texttt{actor='Bill Murray'}}$
scan(Cast)

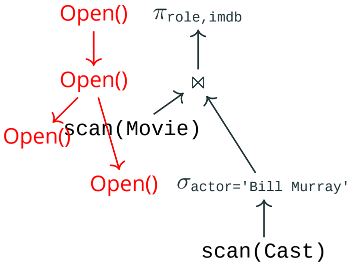
Open()/Close()

Open() is called recursively, top-down, from the root of the tree

A call to **Open()** is supposed to:

(1) create any necessary input buffers
(2) initialize any variables needed (ex: pointers to tuples in a nested loop join operator)
(3) open a file if the operator is a table or index scan

The analogous **Close()** method frees up all those resources.



Open()
$\pi_{\texttt{role,imdb}}$
Open()
$\bowtie$
Open()
scan(Movie)
Open()
$\sigma_{\texttt{actor='Bill Murray'}}$
scan(Cast)

Open()/Close()

Open() is called recursively, top-down, from the root of the tree

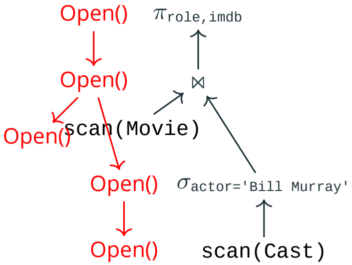
A call to **Open()** is supposed to:

(1) create any necessary input buffers

(2) initialize any variables needed (ex: pointers to tuples in a nested loop join operator)

(3) open a file if the operator is a table or index scan

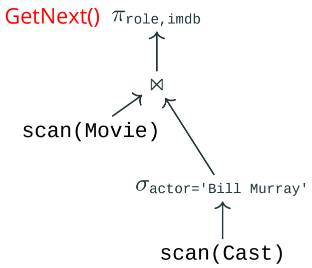
The analogous **Close()** method frees up all those resources.


Open()
$\pi_{\texttt{role,imdb}}$
Open()
$\bowtie$
Open()
scan(Movie)
Open()
$\sigma_{\texttt{actor='Bill Murray'}}$
Open()
scan(Cast)

GetNext()

GetNext() computes one more tuple and returns it to the operator “above” it in the plan.

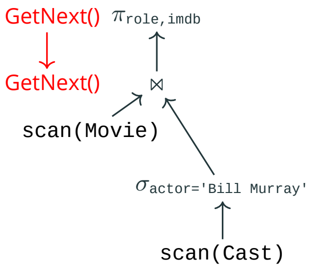
GetNext() returns `<EOF>` when there are no more tuples to return.


GetNext()
$\pi_{\texttt{role,imdb}}$
$\bowtie$
scan(Movie)
$\sigma_{\texttt{actor='Bill Murray'}}$
scan(Cast)

GetNext()

GetNext() computes one more tuple and returns it to the operator "above" it in the plan.

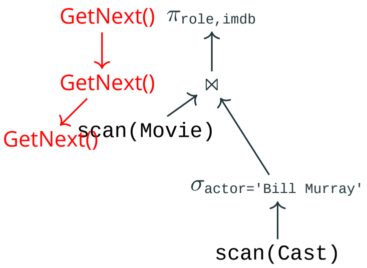
GetNext() returns `<EOF>` when there are no more tuples to return.


GetNext()
$\pi_{\texttt{role,imdb}}$
GetNext()
$\bowtie$
scan(Movie)
$\sigma_{\texttt{actor='Bill Murray'}}$
scan(Cast)

GetNext()

GetNext() computes one more tuple and returns it to the operator “above” it in the plan.

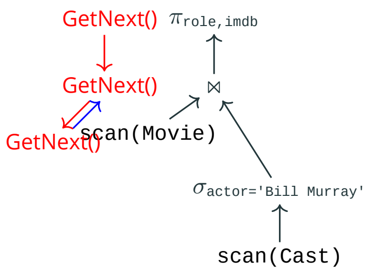
GetNext() returns `<EOF>` when there are no more tuples to return.


GetNext()
$\pi_{\texttt{role,imdb}}$
GetNext()
$\bowtie$
GetNext()
scan(Movie)
$\sigma_{\texttt{actor='Bill Murray'}}$
scan(Cast)

GetNext()

GetNext() computes one more tuple and returns it to the operator “above” it in the plan.

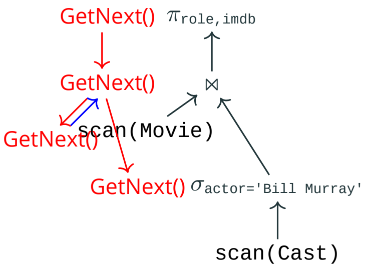
GetNext() returns `<EOF>` when there are no more tuples to return.


GetNext()
$\pi_{\texttt{role,imdb}}$
GetNext()
$\bowtie$
GetNext()
scan(Movie)
$\sigma_{\texttt{actor='Bill Murray'}}$
scan(Cast)

GetNext()

GetNext() computes one more tuple and returns it to the operator “above” it in the plan.

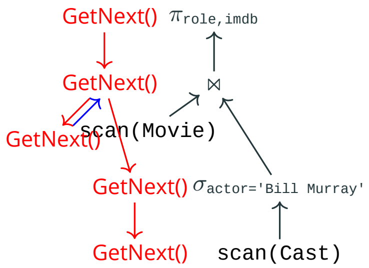
GetNext() returns `<EOF>` when there are no more tuples to return.



GetNext()

GetNext() computes one more tuple and returns it to the operator "above" it in the plan.

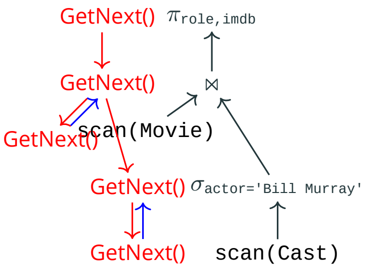
GetNext() returns `<EOF>` when there are no more tuples to return.



GetNext()

GetNext() computes one more tuple and returns it to the operator "above" it in the plan.

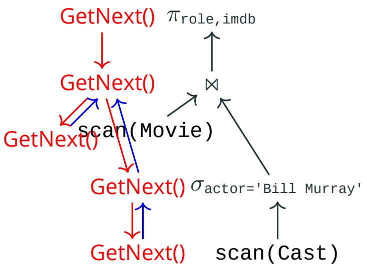
GetNext() returns `<EOF>` when there are no more tuples to return.



GetNext()

GetNext() computes one more tuple and returns it to the operator "above" it in the plan.

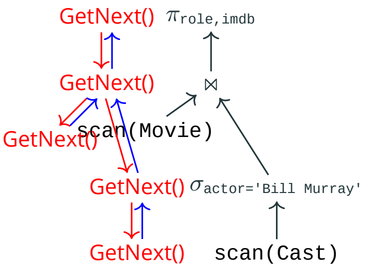
GetNext() returns `<EOF>` when there are no more tuples to return.



GetNext()

GetNext() computes one more tuple and returns it to the operator "above" it in the plan.

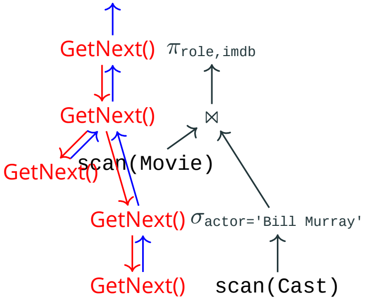
GetNext() returns `<EOF>` when there are no more tuples to return.



GetNext()

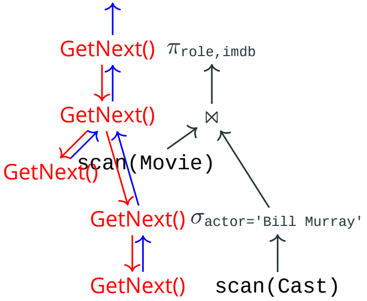
GetNext() computes one more tuple and returns it to the operator “above” it in the plan.

GetNext() returns `<EOF>` when there are no more tuples to return.



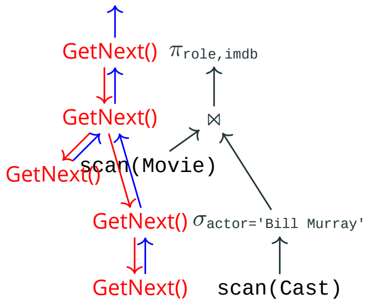
scan(R): returns next tuple from R, reading a new block from disk if needed.

$\pi_{a_1,\ldots,a_n}(R)$: calls `getNext()` on operator below, strip unwanted attributes and return the resulting tuple



$\sigma_C(R)$: repeatedly calls `getNext()` on operator below until a tuple that satisfies the condition C is found; returning that tuple (or `<EOF>` if no such tuple exists).

$R \bowtie_C S$: calls `getNext()` on iterators for R and S according to the algorithm, until a match is found.



Iterators for other binary operators

The iterators for $R - S$, $R \cup S$ and $R \cap S$ are analogous to $R \bowtie_C S$ and make repeated calls to `getNext()` on the iterators for R and S to obtain tuples from each sub-expression.

We will look at intricacies of the algorithms later in these slides.

The next slides show naive (and incomplete) Python-like implementations of some of the iterators, focusing on semantics[3] but omitting details.

Assume there is an interface, called `Iterator`, declaring all operations that each iterator must implement.

- For simplicity, we omit details like constructors.
- `Iterator.from(X)` compiles relational expression `X` into the appropriate iterator object.

[3] After we look at the I/O cost analysis of simple query plans, we will discuss better algorithms for the iterators.

Iterator for $\sigma_C(R)$

```
Class Selection(Iterator):

  def Open():
    self.i = Iterator.from(self.R)
    self.i.Open()


  def Close():
    self.i.Close()
```

```
  def GetNext():
    # read tuple from R
    t = self.i.GetNext()
    while t !=  EOF and
        not satisfies(t, self.C):
      t = i.GetNext()
    yield t
```

The selection condition (`self.c`) is an instance variable[4] because each selection in the plan has its own condition.
Also, `self.i` refers to another iterator, corresponding to the sub-plan “below” that selection.

[4] https://www.digitalocean.com/community/tutorials/understanding-class-and-instance-variables-in-python-3

Iterator for $\pi_{a_1,\ldots,a_n}(R)$

```
Class Projection(Iterator):

  def Open():
    self.i = Iterator.from(self.R)
    self.i.Open()


  def Close():
    self.i.Close()
```

```
def GetNext():
  # read tuple from R
  t = self.i.GetNext()
  if t == EOF:
    yield EOF
  else:
    # remove unwanted attributes
    yield (t.a1, ... , t.an)
```

Iterator for `scan(R)` on a table file

```
Class Scan(Iterator):

  def Open():
    self.F = File.open(self.R)
    self.b = F.first_block()
    self.t = b.first_tuple()


  def Close():
    self.F.Close()
```

```
def GetNext():
  if self.t is None:
    self.b = self.F.next_block()
    if self.b is None:
      yield EOF
    else:
      self.t = self.b.first_tuple()

  old = self.t
  self.t = self.b.next_tuple()
  yield old
```

The state of a table scan

`self.b` and `self.t` represent the **state** of the iterator, and always refer to the tuple that should be returned next.

Iterator for bag version of $R \cup S$ (UNION ALL)

```
Class Union_All(Iterator):

  def Open():
    self.i = Iterator.from(self.R)
    self.i.Open()
    self.reading_R = true


  def Close():
    self.i.Close()

  def GetNext():
    t = self.i.GetNext()
    if t == EOF:
      if reading_R:
        self.i.close()
        self.i = Iterator.from(self.S)
        self.i.Open()
        yield self.i.GetNext()
    else:
      yield EOF
```

No need for duplicate detection

Return all of R first, then all of S (`self.reading_R` tells which one we're reading from).

Iterator for $R \times S$ using nested loops

```
Class Cross_Product(Iterator):

  def Open():
    self.i = Iterator.from(self.R)
    self.i.Open()
    self.j = Iterator.from(self.S)
    self.j.Open()
    self.t_i = self.i.GetNext()


  def Close():
    self.i.Close()
    self.j.Close()
```

```
  def GetNext():
    if self.t_i != EOF:
      t_j = self.j.GetNext()
      if t_j != EOF:
        t = concatenate(self.t_i, t_j)
        yield t
      else:
        self.t_i = self.i.GetNext()
        Iterator.rewind(self.j)
        yield this.GetNext()
    else:
      yield EOF
```

Rewind?

Every time we advance to a new tuple of R we rewind the iterator on S back to the start!

Iterator for $R \bowtie_C S$ using nested loops

```
Class Nested_Loop_Join(Iterator):

  def Open():
    self.i = Iterator.from(self.R)
    self.i.Open()
    self.j = Iterator.from(self.S)
    self.j.Open()
    self.t_i = self.i.GetNext()


  def Close():
    self.i.Close()
    self.j.Close()
```

```
def GetNext():
  if self.t_i != EOF:
    t_j = self.j.GetNext()
    if t_j != EOF:
      t = join(self.t_i, t_j)
      if satisfies(t, self.C):
        yield t
      else:
        yield this.GetNext()
    else:
      self.t_i = self.i.GetNext()
      Iterator.rewind(self.j)
      yield this.GetNext()
  else:
    yield EOF
```

Iterator for `distinct(R)` (duplicate elimination)

```
Class Distinct(Iterator):

  def Open():
    self.i = Iterator.from(self.R)
    self.s = set() # hash set
    self.i.Open()

  def Close():
    self.i.Close()
    self.s = None
```

```
  def GetNext():
    # read tuple from R
    t = self.i.GetNext()
    while t != EOF and t in self.s:
      t = self.i.GetNext()
    yield t
```

Duplicate elimination with an in-memory set

To make sure we never repeat a tuple, we can keep every tuple that is ever returned in a set (`self.s`) in memory.

Implementation considerations

Again, the previous slides are meant to give you **the intuition** of the code for an iterator.

There are many practical considerations that are too low level to list in the notes. For example, for the `distinct()` iterator (slide 33), the DBMS may have to keep the a hash set on disk, depending on the system load at query execution time!

Similarly for joins (and most other binary operators), the DBMS chooses the actual algorithm and data structure to use in real-time, based on the resources (especially RAM) available.

Iterators for Main Memory

Cost model for query processing

Because I/O operations are much more costly than CPU operations, we are not concerned with algorithmic cost (as in CMPUT204).

Instead, we measure the cost of a query plan by:
(1) the number of I/O operations and
(2) the number of memory buffers needed to execute it.[5]

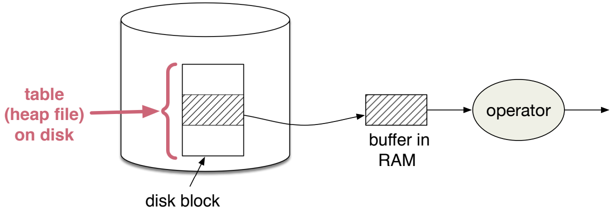
Assumptions

- Each memory buffer holds **one** disk block
- Each I/O operation reads/writes a single block/buffer at a time
- The size of a database file is measured in blocks

[5] The more memory a plan requires the more likely it is to incur costly I/O operations due to swapping.

Table scans

The **table scan** is the simplest way to get tuples from a table. It goes over every tuple in every block (recall slide 29).



A table scan requires only one buffer of RAM (which can be reused). But its I/O cost is $O(|R|)$ (as it goes over every disk block).

How do we find the cost of a plan?

Focus on the iterators that perform I/O or store data in memory.

$$\begin{tikzcd}
{} \\
\pi_{a_1,\ldots,a_n} \arrow[u] \\
\sigma_C \arrow[u] \\
\text{scan}(R) \arrow[u]
\end{tikzcd}$$

Recall that the iterators for σ and π do not read any data from disk, thus have 0 I/O cost.

Also, they do not need to keep any data in buffers. So their memory cost is also 0.

So the final cost of the plan is

$$\boxed{O(1) \text{ buffers in RAM}} \text{ and } \boxed{O(|R|) \text{ I/O operations}}$$

How many buffers should the DBMS use?

A complete table scan can be done with a single buffer.

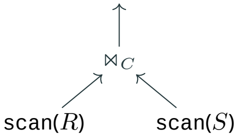
But... IF there is more RAM available, should the DBMS read more blocks at once (maybe all of them) OR ... should it save RAM?

Reading many consecutive blocks at once is better:
- this amortizes the seek time (for HDDs) and
- reduces the impact of BUS congestion

The I/O cost of joins and products

To find out **the actual I/O cost** of the nested-loop-join algorithm of slide 32, consider what happens when we take the calls to `GetNext()` in table scans into account:

```
for each buffer $b_R$ of $R$ do
    for each buffer $b_S$ of $S$ do
        for each tuple $t_R$ in $b_R$ do
            for each tuple $t_S$ in $b_S$ do
                $t' \leftarrow$ join$(t_R, t_S)$
                if $t'$ satisfies $C$ then
                    return $t'$
```



So, how many block reads are done?

Case 1: use <u>one</u> buffer for each scan.

Memory Cost = 2 buffers

I/O Cost =

```
**for each** buffer $b_R$ of $R$ **do**
    **for each** buffer $b_S$ of $S$ **do**
        **for each** tuple $t_R$ in $b_R$ **do**
            **for each** tuple $t_S$ in $b_S$ **do**
                $t' \leftarrow$ join$(t_R, t_S)$
                **if** $t'$ satisfies $C$ **then**
                    **return** $t'$
```

Case 1: use <u>one</u> buffer for each scan.

Memory Cost = 2 buffers

I/O Cost = $\boxed{O(|R| \cdot |S|) \text{ block reads}}$

```
**for each** buffer $b_R$ of $R$ **do**
    **for each** buffer $b_S$ of $S$ **do**
        **for each** tuple $t_R$ in $b_R$ **do**
            **for each** tuple $t_S$ in $b_S$ **do**
                $t' \leftarrow$ join$(t_R, t_S)$
                **if** $t'$ satisfies $C$ **then**
                    **return** $t'$
```

This is the worst case scenario for I/O: quadratic number of block reads.

Case 2: we have M buffers available and $|S| < M - 1$.

```
load all blocks of $S$ to RAM
for each buffer $b_R$ of $R$ do
    for each buffer $b_S$ of $S$ do
        for each tuple $t_R$ in $b_R$ do
            for each tuple $t_S$ in $b_S$ do
                $t' \leftarrow$ join$(t_R, t_S)$
                if $t'$ satisfies $C$ then
                    return $t'$
```

The best case here is to use one buffer to scan for R, and to **load all blocks of S** to buffers in RAM beforehand.

Note that no tuple of either table is read from disk more than once.

Memory Cost = and I/O Cost =

```
load all blocks of $S$ to RAM
for each buffer $b_R$ of $R$ do
    for each buffer $b_S$ of $S$ do
        for each tuple $t_R$ in $b_R$ do
            for each tuple $t_S$ in $b_S$ do
                $t' \leftarrow$ join$(t_R, t_S)$
                if $t'$ satisfies $C$ then
                    return $t'$
```

Case 2: we have M buffers available and $|S| < M - 1$.

The best case here is to use one buffer to scan for R, and to **load all blocks of S** to buffers in RAM beforehand.

Note that no tuple of either table is read from disk more than once.

Memory Cost = $\boxed{|S| + 1}$ and I/O Cost =

Case 2: we have M buffers available and $|S| < M - 1$.

```
load all blocks of $S$ to RAM
for each buffer $b_R$ of $R$ do
    for each buffer $b_S$ of $S$ do
        for each tuple $t_R$ in $b_R$ do
            for each tuple $t_S$ in $b_S$ do
                $t' \leftarrow$ join$(t_R, t_S)$
                if $t'$ satisfies $C$ then
                    return $t'$
```

The best case here is to use one buffer to scan for R, and to **load all blocks of S** to buffers in RAM beforehand.

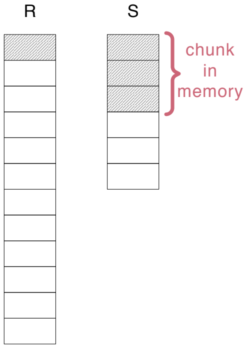
Note that no tuple of either table is read from disk more than once.

Memory Cost = $\boxed{|S| + 1}$ and I/O Cost = $\boxed{|R| + |S|}$

Case 3: M buffers available and $M < |S| \leq |R|$.

Now the best way is to break the inner table into *chunks* of $M - 1$ blocks, and make multiple passes on R using a single buffer:

```
for each chunk of $M - 1$ blocks of $S$ do
    for each buffer $b_R$ of $R$ do
        for each tuple $t_R$ in $b_R$ do
            for each buffer $b_S$ of $S$ in memory do
                for each tuple $t_S$ in $b_S$ do
                    $t' \leftarrow$ join$(t_R, t_S)$
                    if $t'$ satisfies $C$ then
                        return $t'$
```



Cost:

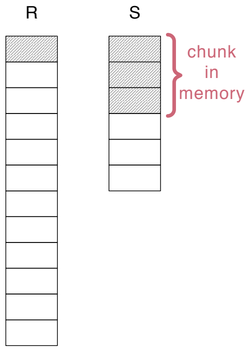
I/O: $\left\lceil \frac{|S|}{M-1} \right\rceil (|R|) + |S|$ block reads

Memory: M buffers

Example:

- $|R|$ = 12 disk blocks
- $|S|$ = 6 disk blocks
- $M = 4$ memory buffers available

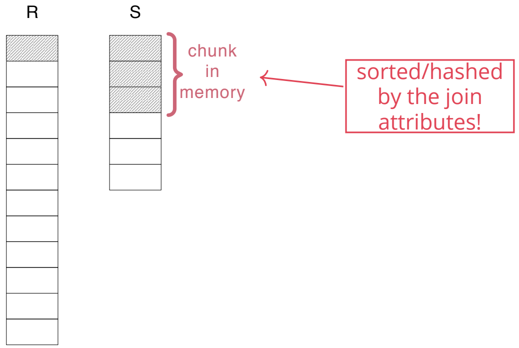
$\left\lceil \frac{6}{3} \right\rceil = 2$ chunks;
total cost = $(2 \cdot 12) + 6 = 30$ block reads.



Speeding up the nested loop join

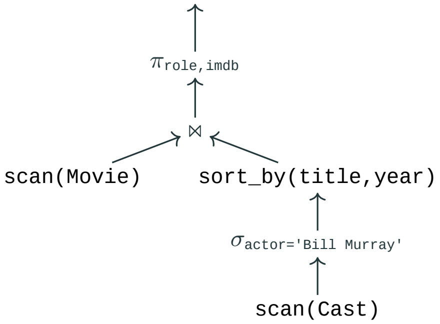
To find matching pairs of tuples faster, the DBMS may sort the tuples of S in memory (by the join attributes).

If the join is based on an equality (`S.a = R.b`), the DBMS may build a hash table with the tuples of S (key: `S.a`, value: whole tuple).



Example: sorting the results of a sub-expression.

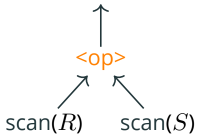
```
SELECT role, imdb
FROM Movie JOIN Cast
WHERE actor='Bill Murray'
```



Better yet, we could sort the results of **both** sub-expressions on the join attributes and perform a sort-based-join (see slide 58) in memory.

Computing set operations with sorting/hashing

Set Union $R \cup S$:



Assumptions:

- M buffers available
- $|R| > |S|$ and $|R \cup S| \leq M - 1$

(1) Keep a hash set H_S in memory.

(2) Read each block of S using a single buffer b_S;

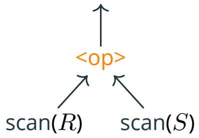
- For each t_S in b_S: if it is not in H_S, add it to H_S and return it to the caller.

(3) Read each block of R using a single buffer b_R;

- For each t in b_R: if it is not in H_S, add it to H_S and return it to the caller.

Set Intersection $R \cap S$:

Assumptions:
- M buffers available
- $|R| > |S|$ and $|S| \leq M - 1$

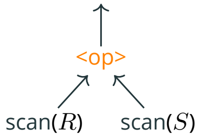


(1) Read all of S into a hash set (size $M - 1$ buffers) in memory.

(2) Read each block of R to (single) buffer b_R;

- For each t in b_R: if it is in H_S, remove it from H_S and return it to the caller.

Set Difference $R - S$ (Algorithm 1):



Assumptions:

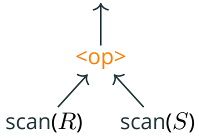
- M buffers available
- $|R| > |S|$ and $|S \cup (R - S)| \leq M - 1$

(1) Read all of S into a hash set H_S in memory.

(2) Read each block of R to (single) buffer b_R;

- For each t in b_R: if it is not in H_S, $\underline{\text{add it to } H_S}$ and return it to the caller.

Set Difference $R - S$ (Algorithm 2)



Assumptions:

- M buffers available
- $|S| > |R|$ and $|R| \leq M - 1$

(1) Read all of R into a hash set H_R in memory.

(2) Read each block of S to (single) buffer b_S;

- For each t_S in b_S: if t_S **is** in H_R remove it from H_R.

(3) Return the tuples in H_R (one at a time) to the caller.

Duplicate elimination

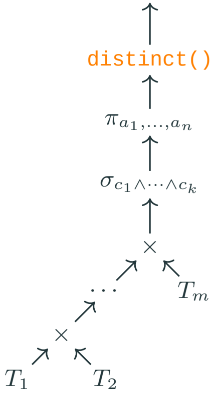
Removing duplicates is done last in the plan!

Idea: keep the tuples from the query (Q) in a hash set, using $\underline{\text{up to}}$ M buffers; discard duplicates as they arrive (see slide 33).

```
SELECT DISTINCT a₁, ... , aₙ
FROM T₁ [, T₂ [, ... [, Tₘ]]]
WHERE c₁ AND c₂ AND … AND cₖ
```

Assumption: $|Q| < M$

Cost = $\boxed{\text{0 I/O}}$ and $\boxed{O(|Q|) \text{ buffers.}}$



Iterators for External Memory

If the tables are too large to fit in memory, a good strategy to join them is to create sorted copies of the tables on the respective join attributes and scan these tuples in sorted order.

Example: $R \bowtie_{R.a=S.b} S$ when $|R| >> |S| >> M$ requires:

- Copy of R, sorted by R.a
- Copy of S, sorted by S.b

The DBMS **must make a copy** of the table, instead of sorting the original table file.

- This allows other queries (and even updates) on that table to be executed while the sorting goes on.

Multi-way Merge Sort on external memory:

Assume there are M buffers available and you need to sort a file with $|R| >> M$ blocks.

Let $N = \left\lceil \frac{|R|}{M-1} \right\rceil$.

(1) Load “chunks” of R into $M-1$ buffers in memory, sort the tuples, and write the buffers, one buffer at a time, to disk.

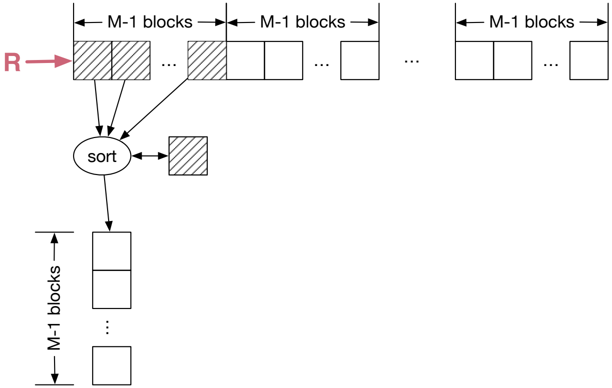
(2) Merge the N sorted chunks (assuming $N \leq M-1$):

  (a) Write the output to 1 buffer in memory; flush it to disk when full

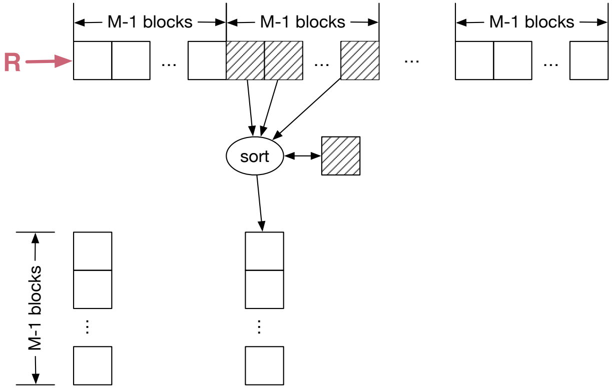
  (b) Read the N sorted chunks using 1 buffer per file

  (c) Compare the “current” tuples in each buffer; add the “smallest” to the output buffer; advance that pointer

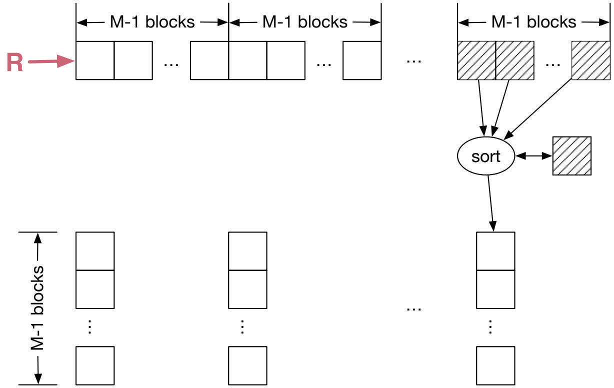
Step 1: Sort N individual "chunks" of the table in memory, and write them back to disk.



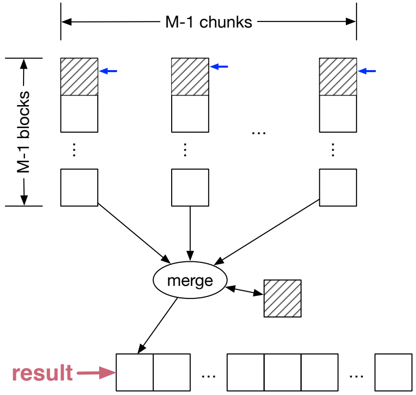
Step 1: Sort N individual “chunks” of the table in memory, and write them back to disk.



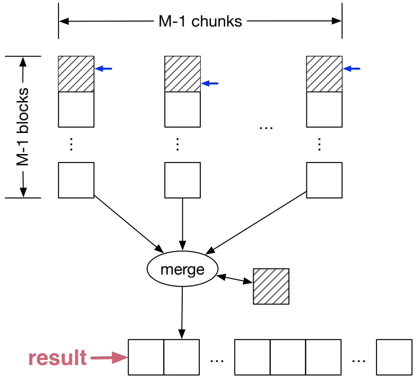
Step 1: Sort N individual "chunks" of the table in memory, and write them back to disk.



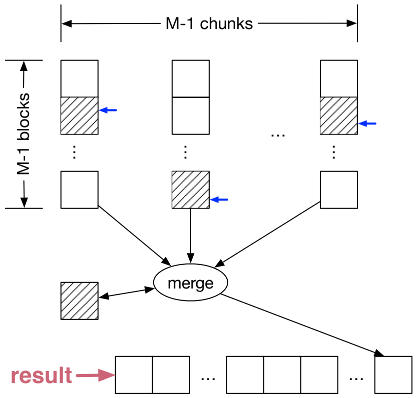
Step 2: Merge the N sorted chunks into a single sorted file. Case when $N \leq M-1$:



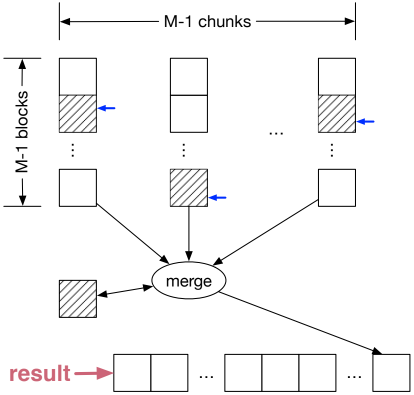
Step 2: Merge the N sorted chunks into a single sorted file. Case when $N \leq M - 1$:



Step 2: Merge the N sorted chunks into a single sorted file. Case when $N \leq M - 1$:

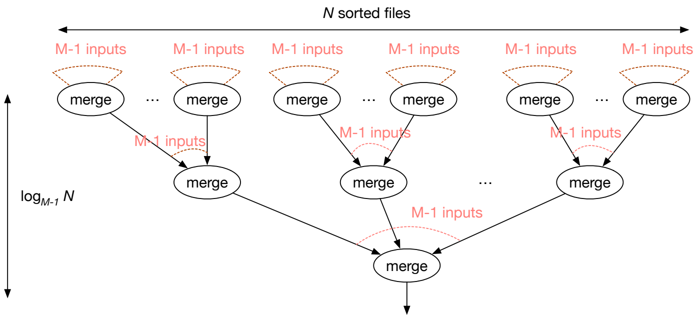


Step 2: Merge the N sorted chunks into a single sorted file. Case when $N \leq M - 1$:



When $N \leq M - 1$ only one merge operation is needed.

Step 2 revisited when $N > M - 1$:



This is like in-RAM mergesort, except that each merge steps has $M - 1$ inputs **instead of** two.

- The tree has $\lceil \log_{M-1} N \rceil$ levels.
- The entire table is read and written in each level, for a total I/O cost of $\boxed{O(|R| \lceil \log_{M-1} N \rceil)}$

External multi-way mergesort:

M : buffers available; $|R|$: blocks to sort; $N = \left\lceil \frac{|R|}{M-1} \right\rceil$: chunks.

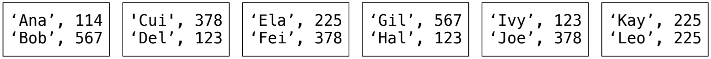
- **Sort phase:** load, sort and write each chunk.
- **Merge phase:** start with $pass = 1$
  (1) Merge the sorted chunks, $M - 1$ at a time.
  (2) If the pass produced 2 or more chunks, increment $pass$ and repeat step (1).

How much I/O in total?

In each pass on the file we read and write the same amount of data (i.e., the size of the file itself):

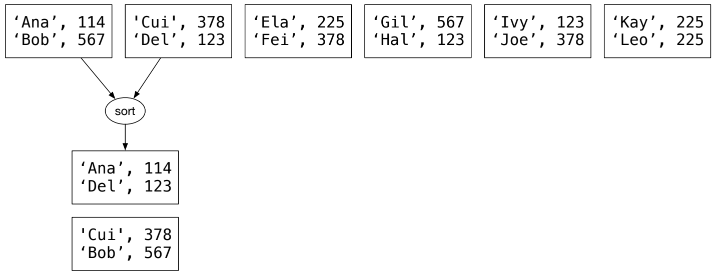
- Sort phase: $2 \cdot |R|$
- Merge phase: $2 \cdot \lceil \log_{M-1} N \rceil |R|$

Example: sorting relation `Member(name, class)` on `class` with $M = 3$.

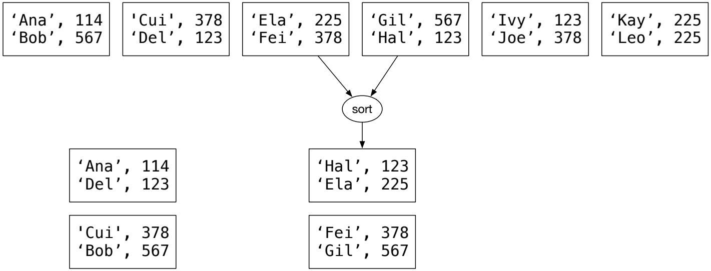


```
‘Ana’, 114   'Cui', 378   ‘Ela’, 225   ‘Gil’, 567   ‘Ivy’, 123   ‘Kay’, 225
‘Bob’, 567   ‘Del’, 123   ‘Fei’, 378   ‘Hal’, 123   ‘Joe’, 378   ‘Leo’, 225
```

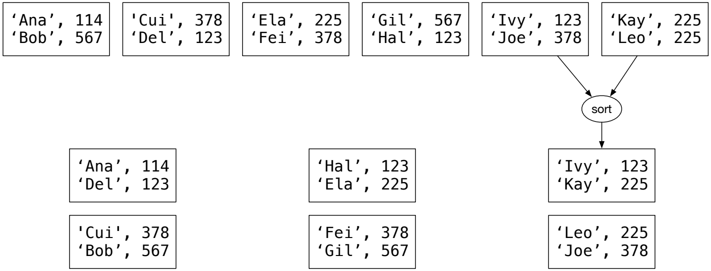
Example: sorting relation `Member(name, class)` on `class` with $M = 3$.



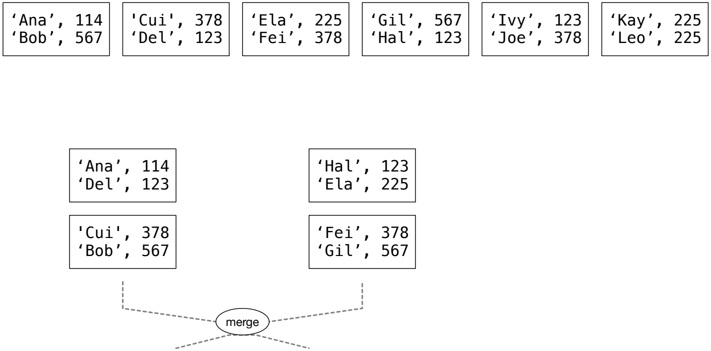
Example: sorting relation `Member(name, class)` on `class` with $M = 3$.



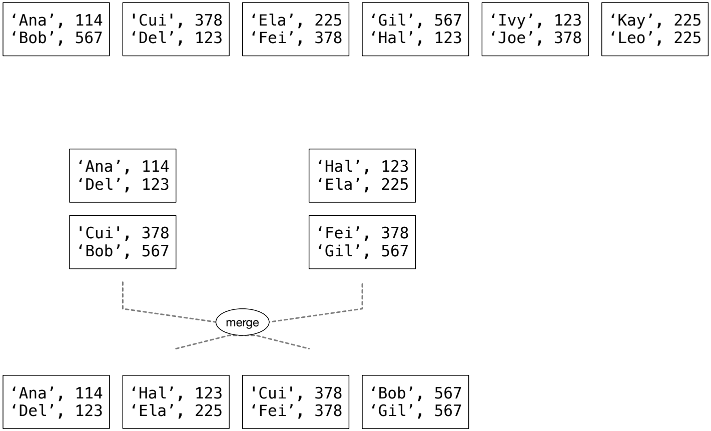
Example: sorting relation `Member(name, class)` on `class` with $M = 3$.



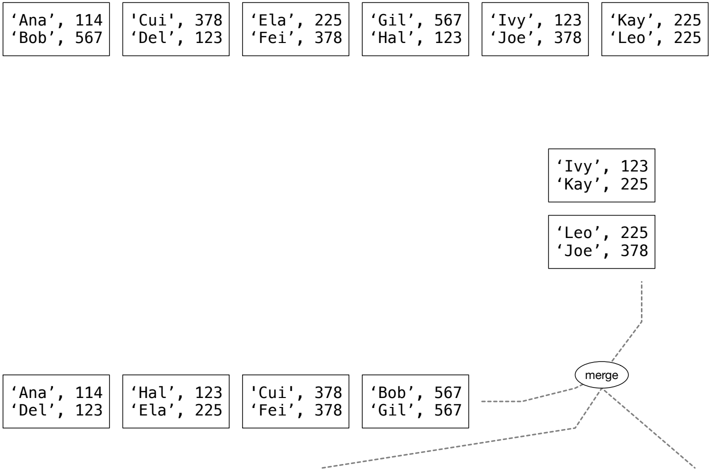
Example: sorting relation `Member(name, class)` on `class` with $M = 3$.



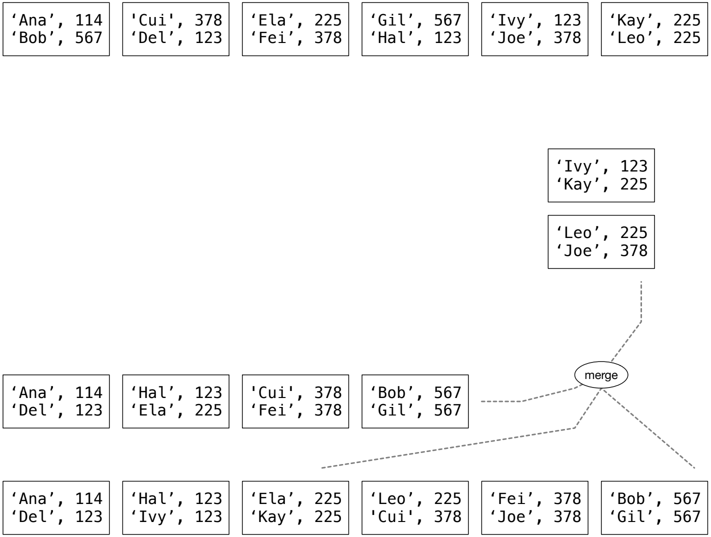
Example: sorting relation `Member(name, class)` on `class` with $M = 3$.



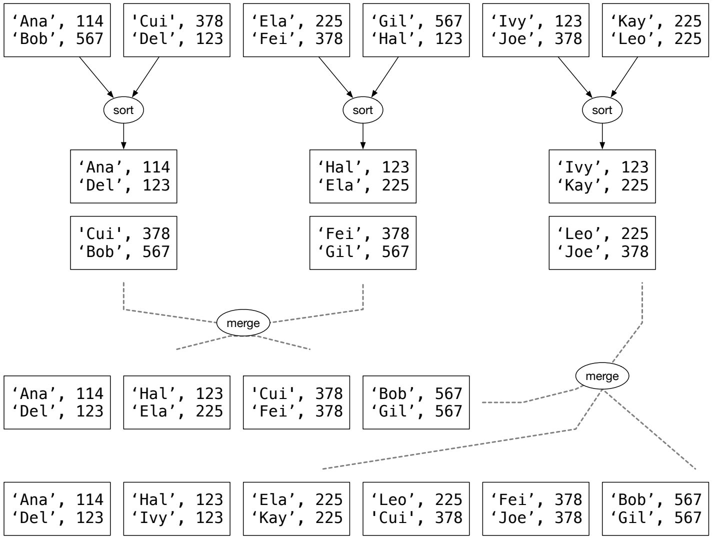
Example: sorting relation `Member(name, class)` on `class` with $M = 3$.



Example: sorting relation `Member(name, class)` on `class` with $M = 3$.



Example: sorting relation `Member(name, class)` on `class` with $M = 3$.



Sort-based Equality Join $R.a = S.b$

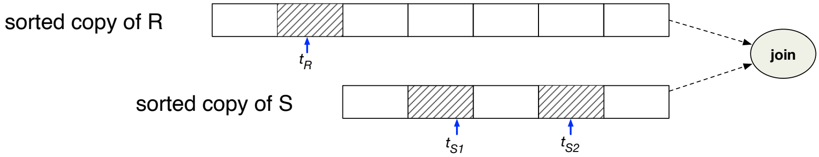
(1) Make a sorted copy of R (on R.a) and another of S (on S.b) using the external merge-sort algorithm.

(2) Scan the two sorted relations looking for tuples that match:

t_R : current tuple of R

t_{S1} : first tuple in S such that $t_R.a = t_{S1}.b$

t_{S2} : current tuple in S such that $t_R.a = t_{S2}.b$



Example: `SELECT name, time FROM Member JOIN Schedule`

Member (name, class)

‘Ana’, 114 ‘Del’, 123	‘Hal’, 123 ‘Ivy’, 123	‘Ela’, 225 ‘Kay’, 225	‘Leo’, 225 'Cui', 378	‘Fei’, 378 ‘Joe’, 378	‘Bob’, 567 ‘Gil’, 567

Schedule (class, time)

114, ’M 8’ 114, ’W 8’	123, ’T 7’ 378, ‘M 6’	378, ’F 6’ 474, ‘W 6’	567, ’R 7’

Example: `SELECT name, time FROM Member JOIN Schedule`

```
Member (name, class)

‘Ana’, 114   ‘Hal’, 123   ‘Ela’, 225   ‘Leo’, 225   ‘Fei’, 378   ‘Bob’, 567
‘Del’, 123   ‘Ivy’, 123   ‘Kay’, 225   'Cui', 378   ‘Joe’, 378   ‘Gil’, 567

Schedule (class, time)

114, ’M 8’   123, ’T 7’   378, ’F 6’   567, ’R 7’
114, ’W 8’   378, ‘M 6’   474, ‘W 6’



‘Ana’,’M 8’
‘Ana’,‘W 8’
```

Example: `SELECT name, time FROM Member JOIN Schedule`

```
Member (name, class)

‘Ana’, 114   ‘Hal’, 123   ‘Ela’, 225   ‘Leo’, 225   ‘Fei’, 378   ‘Bob’, 567
‘Del’, 123   ‘Ivy’, 123   ‘Kay’, 225   'Cui', 378   ‘Joe’, 378   ‘Gil’, 567
```

```
Schedule (class, time)

114, ’M 8’   123, ’T 7’   378, ’F 6’   567, ’R 7’
114, ’W 8’   378, ‘M 6’   474, ‘W 6’
```

```
‘Ana’,’M 8’   ‘Del’,’T 7’   ‘Ivy’,’T 7’
‘Ana’,‘W 8’   ‘Hal’,’T 7’
```

Example: `SELECT name, time FROM Member JOIN Schedule`

Member (name, class)

```
‘Ana’, 114   ‘Hal’, 123   ‘Ela’, 225   ‘Leo’, 225   ‘Fei’, 378   ‘Bob’, 567
‘Del’, 123   ‘Ivy’, 123   ‘Kay’, 225   'Cui', 378   ‘Joe’, 378   ‘Gil’, 567
```

Schedule (class, time)

```
114, ’M 8’   123, ’T 7’   378, ’F 6’   567, ’R 7’
114, ’W 8’   378, ‘M 6’   474, ‘W 6’
```

```
‘Ana’,’M 8’   ‘Del’,’T 7’   ‘Ivy’,’T 7’   ‘Cui’,’F 6’   ‘Fei’,’F 6’   ‘Joe’,’F 6’
‘Ana’,‘W 8’   ‘Hal’,’T 7’   ‘Cui’,’W 6’   ‘Fei’,’M 6’   ‘Joe’,’M 6’
```

Example: `SELECT name, time FROM Member JOIN Schedule`

Member (name, class)

```
‘Ana’, 114   ‘Hal’, 123   ‘Ela’, 225   ‘Leo’, 225   ‘Fei’, 378   ‘Bob’, 567
‘Del’, 123   ‘Ivy’, 123   ‘Kay’, 225   'Cui', 378   ‘Joe’, 378   ‘Gil’, 567
```

Schedule (class, time)

```
114, ’M 8’   123, ’T 7’   378, ’F 6’   567, ’R 7’
114, ’W 8’   378, ‘M 6’   474, ‘W 6’
```

Example: `SELECT name, time FROM Member JOIN Schedule`

`Member (name, class)`

‘Ana’, 114	‘Hal’, 123	‘Ela’, 225	‘Leo’, 225	‘Fei’, 378	‘Bob’, 567
‘Del’, 123	‘Ivy’, 123	‘Kay’, 225	'Cui', 378	‘Joe’, 378	‘Gil’, 567

`Schedule (class, time)`

114, ’M 8’	123, ’T 7’	378, ’F 6’	567, ’R 7’
114, ’W 8’	378, ‘M 6’	474, ‘W 6’	

‘Ana’,’M 8’	‘Del’,’T 7’	‘Ivy’,’T 7’	‘Cui’,’F 6’	‘Fei’,’F 6’	‘Joe’,’F 6’	‘Gil’,’R 7’
‘Ana’,‘W 8’	‘Hal’,’T 7’	‘Cui’,’W 6’	‘Fei’,’M 6’	‘Joe’,’M 6’	‘Bob’,’R 7’	

Pseudo-code of sort-based equality join. For simplicity, we refer to pointers to the tuples inside the sorted files.

Open() for sort-based evaluation of $R \bowtie_{R.a=S.b} S$

```
1: $t_R \leftarrow$ first tuple in $R$; $t_{S1} \leftarrow$ first tuple in $S$; $t_{S2} \leftarrow t_{S1}$
```

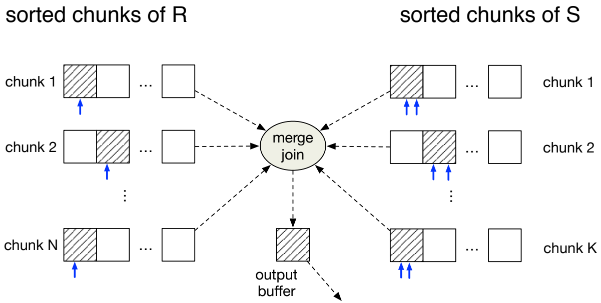
GetNext() for sort-based evaluation of $R \bowtie_{R.a=S.b} S$

```
1: while $t_R.a \neq t_{S1}.b$ do
2:     if$t_R.a < t_{S1}.b$ then advance $t_R$ else advance $t_{S1}$ ; $t_{S2} = t_{S1}$
3: if $t_R =$ EOF then    % stop when there are no more tuples in $R$ to be joined
4:     return EOF
5: if $t_R.a = t_{S2}.b$ then
6:     $t \leftarrow$join($t_R$,$t_{S2}$)
7:     advance $t_{S2}$
8:     if $t_{S2} =$ EOF OR $t_{S2}.b > t_R.a$ then    % all matches of the current $t_R$ found
9:         advance $t_R$    % the next call to GetNext() will find matches of the next $t_R$
10:    yield $t$
```

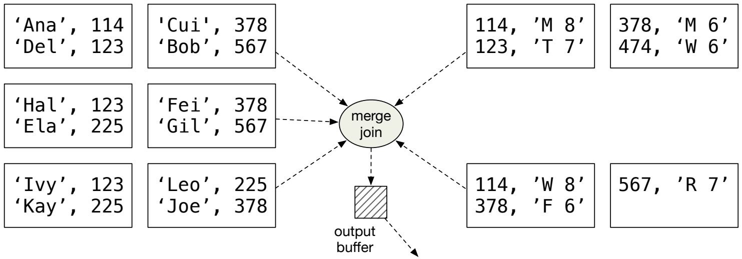
Merge-Join on equality $R.a = S.b$

If there are enough buffers to read all chunks of R and S concurrently, one can perform the join itself during the “merge” phase on the previous algorithm.

Gist: let p point to the “smallest” tuple in R; find all matching tuples in S (as in the previous algorithm); advance p; repeat.



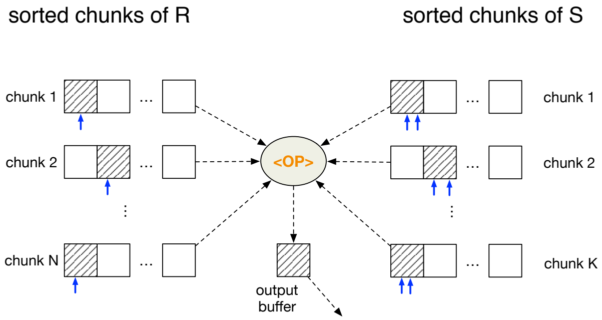
Example of merge-join:



Sort-based algorithms for other binary operators

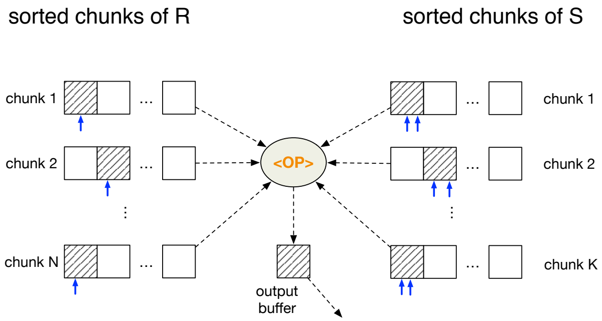
The same “sort-merge” framework can be used for the other binary operators.

Set Union $R \cup S$: copy the next “smallest” tuple amongst all pointers (in either R and S) to the output; advance all pointers to duplicates of that tuple until they point to something else.



Set Intersection $R \cap S$: copy the next "smallest" tuple that appears in both R and S to the output; advance all pointers to duplicates of that tuple until they point to something else.

Set difference $R - S$: copy the next "smallest" tuple in R but not in S (or vice-versa for $S - R$); advance all pointers on R (or S, for $S - R$) until a new "smallest" tuple is found.

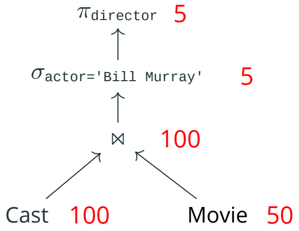


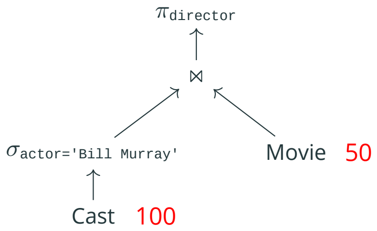
Cost Estimation

Which plan is better?

Consider the two following plans that compute the same answer.

They have the same I/O and memory cost. Which one will take the least **time** to execute?

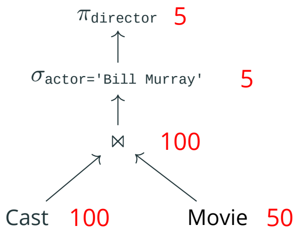




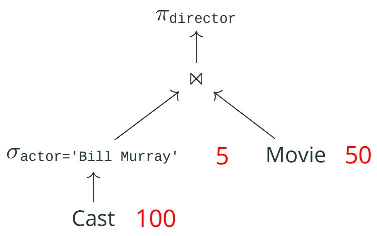
Which plan is better?

Consider the two following plans that compute the same answer.

They have the same I/O and memory cost. Which one will take the least **time** to execute?



$\pi_{\texttt{director}}$
5
$\sigma_{\texttt{actor='Bill Murray'}}$
5
$\bowtie$
100
Cast
100
Movie
50

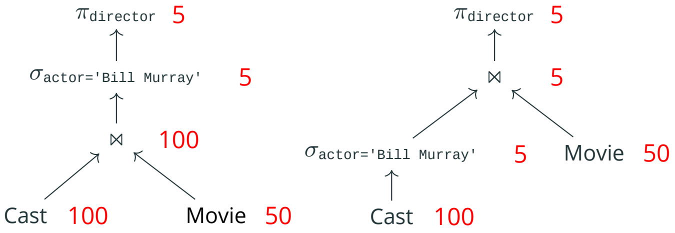


$\pi_{\texttt{director}}$
$\bowtie$
$\sigma_{\texttt{actor='Bill Murray'}}$
5
Movie
50
Cast
100

Which plan is better?

Consider the two following plans that compute the same answer.

They have the same I/O and memory cost. Which one will take the least **time** to execute?


$\pi_{\texttt{director}}$ 5
$\sigma_{\texttt{actor='Bill Murray'}}$ 5
$\bowtie$ 100
Cast 100
Movie 50
$\pi_{\texttt{director}}$ 5
$\bowtie$ 5
$\sigma_{\texttt{actor='Bill Murray'}}$ 5
Movie 50
Cast 100

The previous example seems trivial: the DBMS can never do worse by performing the selection $\sigma_{\texttt{actor='Bill Murray'}}$ before the join.

Sometimes the DBMS faces choices which are far from trivial.

Example 1: if a query joins 3 tables R, S, T, should they be joined like this $(R \bowtie S) \bowtie T$? Or is there a better ordering?

Example 2: if there exists a secondary index on a table, should it be used or should the DBMS scan the table instead?

The next slides look into how the DBMS can choose a plan among several alternatives.

Result sizes and query time

As we just saw, two plans for the same query, with the same I/O and memory cost can have different execution time, depending on the number of tuples that are computed by intermediate results.

However, the DBMS cannot execute various plans to know the exact size of their intermediate results.

Instead, the DBMS needs to **estimate** those sizes for any given plan, using statistics about the database.

Estimated vs Real Cost

Using estimated costs, the DBMS can evaluate many candidate plans in less time than it would take to execute any of them. In practice estimated cost is correlated with actual cost.

Database statistics

The following are the simplest statistics (per relation R in the database) the DBMS can use to estimate the cost of a query plan.

$B(R)$	number of of blocks to hold all tuples in R
$T(R)$	number of tuples in R
$S(R)$	number of of bytes per tuple in R
$V(R, a)$	number of distinct values of a in R

The DBMS might also keep the lowest and the highest values of numeric attributes:

$low(R, a)$	lowest value of a in R
$high(R, a)$	highest value of a in R

Example

```
CREATE TABLE Movie (title CHAR(20), year INT, imdb FLOAT,
   director CHAR(20), PRIMARY KEY (title, year),
   CHECK(imdb >= 0 AND imdb <= 10));
```

Assume:

- `sizeOf(int)` = 4 bytes
- `sizeOf(char)` = 2 bytes
- `sizeOf(float)` = 8 bytes

Movie

title	year	imdb	director
Ghostbusters	1984	7.8	Ivan Reitman
Big	1988	7.3	Penny Marshall
Lost in Translation	2003	7.8	Sofia Coppola
Wadjda	2012	8.1	Haifaa al-Mansour
Ghostbusters	2016	5.3	Paul Feig

$$T(\texttt{Movie}) = 5 \qquad S(\texttt{Movie}) = 92$$

$$V(\texttt{Movie}, \texttt{title}) = 4 \qquad V(\texttt{Movie}, \texttt{year}) = 5 \qquad V(\texttt{Movie}, \texttt{imdb}) = 4 \qquad V(\texttt{Movie}, \texttt{director}) = 5$$

$$low(\texttt{Movie}, \texttt{year}) = 1984 \qquad high(\texttt{Movie}, \texttt{year}) = 2016 \qquad low(\texttt{Movie}, \texttt{imdb}) = 5.3 \qquad high(\texttt{Movie}, \texttt{imdb}) = 8.1$$

Result Size Estimation

To estimate the size of a query or sub-expression Q we need to know $T(Q)$ and $S(Q)$.

Since memory buffers have a fixed size, we can always estimate the number of buffers (and thus how much RAM) is needed as

$$B(Q) = \left\lceil \frac{T(Q) \cdot S(Q)}{\text{buffer size}} \right\rceil$$

For simplicity, we consider here the cases where all tuples are of fixed size. For variable-sized tuples, the DBMS can use the average tuple size, for example, to get a reasonable estimate.

Estimates with Selections

Let $Q : \sigma_{a_i = v}(R)$.

What are good estimates for $S(Q)$ and $T(Q)$ and $V(Q, \cdot)$?

$$\begin{aligned} S(Q) &= S(R) \quad ^{\dagger} \\ V(Q, a_i) &= 1 \quad ^{\ddagger} \\ V(Q, a_j) &= V(R, a_j) \qquad j \neq i \end{aligned}$$

† Since the selection does not change the tuples.

‡ For numeric attributes we can do better by checking if $low(R, a_i) \leq v \leq high(R, a_i)$ and assume 0 otherwise. For non-numeric attributes, such analysis is much harder.

What about $T(Q)$? How many tuples of R should have $a_i = v$?

Predicate selectivity

The selectivity of predicate c_i is defined as

$$s(R, c_i) = \frac{T(\sigma_{c_i}(R))}{T(R)}$$

and estimated as in the previous slides.

In general, $T(\sigma_{c_i}(R)) = \lceil s(R, c_i) \cdot T(R) \rceil$, for any expression R (not just base tables).

Estimating predicate selectivity:

We need to make assumptions about how the values are distributed in order to make any estimates.

One common assumption in cost estimation is that the values are **uniformly distributed** across the table, meaning that, for any given attribute a_i:

- All $V(R, a_i)$ distinct values appear the same number of times.
- All values are *equally likely* to appear in any given tuple t.

Thus: $$\boxed{s(R, \sigma_{a_i=v}) = \frac{1}{V(R,a_i)}; \qquad T(Q) = \left\lceil \frac{1}{V(R,a_i)} \cdot T(R) \right\rceil}.$$

An alternative estimate is to assume all values in the domain are equally likely. This, of course, only makes sense for attributes with a discrete domain.

In this case, we'd have $s(R, a_i = v) = \frac{1}{|\text{domain}(a_i)|}$

Example:

```
CREATE TABLE Instructor(name TEXT, faculty TEXT, email TEXT,
    CHECK (faculty IN ('Arts','Science','Engineering',
                       'Business','Law','Medicine')));
```

$$T(\sigma_{\texttt{faculty='Arts'}}(\texttt{Instructor})) = \left\lceil \frac{1}{6} \cdot T(\texttt{Instructor}) \right\rceil$$

The estimate is always the same for values that belong to the domain, and 0 for values not in the domain.

Selections With Ranges

When the selection in based on a range of values $Q = \sigma_{a_i > v}(R)$ the estimates for $T(Q)$ and $V(Q, a_i)$ need to be adjusted.

With *low*/*high* statistics, we can make some reasonable estimates. If $low(R, a_i) \leq v \leq high(R, a_i)$[6], we can use:

$$T(Q) = \left\lceil \frac{high(R, a_i) - v}{high(R, a_i) - low(R, a_i)} T(R) \right\rceil \text{ and}$$

$$V(Q, a_i) = \left\lceil \frac{high(R, a_i) - v}{high(R, a_i) - low(R, a_i)} V(R, a_i) \right\rceil$$

Estimates for $Q = \sigma_{a_i < v}(R)$ are computed analogously.

[6] The boundary cases when v is out of that range are left as an exercise.

Example: $Q = \sigma_{\texttt{imdb>7}}(\texttt{Movie})$.

$$T(Q) = \left\lceil \frac{8.1 - 7}{8.1 - 5.3}\, T(\texttt{Movie}) \right\rceil = \lceil 1.96 \rceil = 2$$

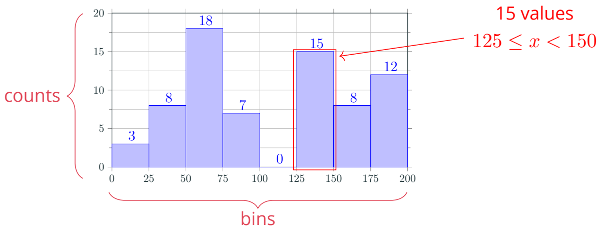
$$V(Q, \texttt{imdb}) = \left\lceil \frac{8.1 - 7}{8.1 - 5.3}\, V(\texttt{Movie}, \texttt{imdb}) \right\rceil = \lceil 1.57 \rceil = 2$$

These estimates are not great because the imdb values are **not** uniformly distributed. We would need better statistics to be more accurate here.

Better estimations with histograms

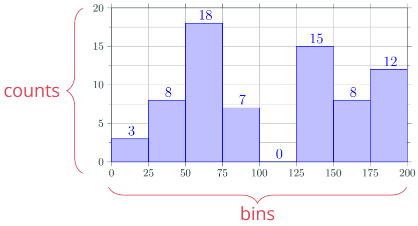
Histograms are statistical summaries of data that record the number of values that fall within predefined **bins**.

In an *equi-width* histogram, all bins have the same width, and cover the same number of elements in the domain.



We can estimate the the probability of a tuple satisfying a selection condition with a histogram of the attribute in the selection.

$$s(R, a_i = x) = \frac{count(bin(x))}{T(R)}$$



Examples:

- $s(R, a_i = 180) = 12/71$
- $s(R, a_i = 60) = 18/71$
- $s(R, a_i = 110) = 0$

For ranges, sum-up across bins overlapping with the range:

- $s(R, a_i \geq 100) = (0 + 15 + 8 + 12)/71$
- $s(R, 10 \leq a_i < 50) = (3 + 8)/71$

For $V(Q, a_i)$ we can use either the number of values in the range (as before) or the sum of counts in the respective bins, whichever is smaller.

Histograms are an example of how better statistics offer more accurate predictions.

The trade-off is keeping the histogram up-to-date.

- Update the histogram after every database update **or** only when the database goes offline for maintenance?

What about textual attributes?

Histograms defined by equally-spaced strings (e.g., "A", "C", "E", ...) could help with equality queries... but not *wild-card queries*:

```
SELECT title, imdb
FROM Movie
WHERE title LIKE "%space%"
```

Getting accurate estimates is not easy!

What about $Q = \sigma_{c_1 \wedge \cdots \wedge c_n}(R)$?

How many tuples satisfy all of the conditions at the same time? This can be estimated with a bit of probability theory.

Using $s(R, c_i)$ as the probability that a tuple in R satisfies c_i, and assuming attribute values are *independent* of each other, we have

$$T(\sigma_{c_1 \wedge \cdots \wedge c_n}(R)) = \left\lceil \left(\prod_{1 \le i \le n} s(R, c_i) \right) T(R) \right\rceil$$

Example: $Q = \sigma_{\texttt{imdb>7} \ \wedge \ \texttt{year=2016}}(\texttt{Movie})$.

$$T(Q) = \left\lceil \left(\frac{2}{5} \cdot \frac{1}{5} \right) \cdot 5 \right\rceil = \left\lceil \frac{2}{5} \right\rceil = 1$$

Also using $s(R, c_i)$ as the probability that a tuple in R satisfies c_i, and assuming attribute values are *independent* of each other, we can estimate the size of a disjunction as:

$$T(\sigma_{c_1 \vee \ldots \vee c_n}(R)) = \left\lceil \left(1 - \prod_{1 \leq i \leq n} 1 - s(R, c_i)\right) T(R) \right\rceil$$

The double negation computes the probably that at least one c_i is satisfied as the complement of the probability that *none* is.

Example: $Q = \sigma_{\texttt{imdb>7} \ \vee \ \texttt{year=2016}}(\texttt{Movie})$.

$$T(Q) = \left\lceil \left(1 - \left(\frac{3}{5} \cdot \frac{4}{5}\right)\right) \cdot 5 \right\rceil = \left\lceil \frac{65}{25} \right\rceil = 3$$

Estimates with Projections

If $Q = \pi_{a_1,\ldots,a_n}(R)$, what are good estimates for $S(Q), T(Q)$ and $V(Q,\cdot)$?

$$\begin{aligned} T(Q) &= T(R) \quad ^{\dagger} \\ S(Q) &= \texttt{sizeOf}(a_1) + \cdots + \texttt{sizeOf}(a_n) \\ V(Q, a_i) &= V(R, a_i) \;\; \text{if } a_i \text{ in project list} \end{aligned}$$

† The projection operator in SQL does not remove duplicates.

Estimates with Cartesian Products

What about $Q = R_1 \times R_2$?

$$\begin{aligned} T(Q) =& T(R_1)T(R_2) \quad ^{\dagger} \\ S(Q) =& S(R_1) + S(R_2) \quad ^{\ddagger} \end{aligned}$$

$$V(Q, a_i) = \begin{cases} V(R_1, a_i) & \text{if } a_i \in R_1 \\ V(R_2, a_i) & \text{if } a_i \in R_2 \end{cases}$$

† the Cartesian product produces **all pairings** of tuples.

‡ the Cartesian product **concatenates** tuples together.

Estimates with Joins

Case #1: A natural join $Q = R_1 \bowtie R_2$ where R_1 and R_2 share a common attribute a_i.

We immediately know that $\boxed{S(Q) = S(R_1) + S(R_2) - \texttt{sizeOf}(a_i)}$

We also know that $\boxed{0 \leq T(Q) \leq T(R_1)T(R_2)}$... but can we get a better estimate?

If $R_1.a_i$ is the primary key of R_1 and $R_2.a_i$ is a foreign key referencing R_1, we know that every tuple in R_2 can match at most one tuple in R_1, resulting in that $T(Q) = T(R_2)$.

What if a_i is not a key of either relation? We need to make assumptions to estimate $T(Q)$ and $V(Q, \cdot)$.

We assume that either $R_1.a_i \subseteq R_2.a_i$ or $R_2.a_i \subseteq R_1.a_i$. That is, all values of the join attribute in the relation with the least unique values appear in the other relation[7].

Then, $\boxed{V(Q, a_i) = \min\left(V(R_1, a_i), V(R_2, a_i)\right)}$.

We also assume that the number of values among attributes $a_j \neq a_i$ not involved in the join are *preserved*:

$$V(Q, a_j) = \begin{cases} V(R_1, a_j) & \text{if } a_j \in R_1 \\ V(R_2, a_j) & \text{if } a_j \in R_2 \end{cases}$$

[7]This is called the **Containment of Value Sets** Assumption.

To estimate $T(R_1 \bowtie R_2)$, consider first:

How many tuples of R_2 will match a tuple $t \in R_1$?

This is the same as estimating $T(\sigma_{a_i=v}(R_2))$ where $v = t.a_i$.

Since $T(\sigma_{a_i=t.a_i}(R_2)) = \left\lceil \frac{1}{V(R_2,a_i)} \cdot T(R_2) \right\rceil$, and since we assumed that every tuple in R_1 will match at least one tuple in R_2, we have:

$$T(R_1 \bowtie R_2) = \left\lceil T(R_1) \cdot \frac{T(R_2)}{V(R_2, a_i)} \right\rceil$$

Swapping R_1 and R_2 in the analysis above gives another estimate:

$$T(R_1 \bowtie R_2) = \left\lceil T(R_2) \cdot \frac{T(R_1)}{V(R_1, a_i)} \right\rceil$$

If the two estimates disagree, we take the least of them, arriving at

$$T(Q) = \left\lceil \frac{T(R_1)T(R_2)}{\max(V(R_1, a_i), V(R_2, a_i))} \right\rceil$$

Because a join is equivalent to a Cartesian product followed by a selection

$$R_1 \bowtie_c R_2 = \sigma_c(R_1 \times R_2)$$

And because we assume that $T(\sigma_c(R) = \lceil s(R, c)T(R) \rceil$, we arrive at an expression for the selectivity of an equality predicate in a join:

$$s(R_1 \times R_2, R_1.a_i = R_2.a_i) = \frac{1}{\max(V(R_1, a_i), V(R_2, a_i))}$$

Let $Q = R_1(a, b, c) \bowtie_{a=d} R_2(d, e)$.

In this case, note that there are no attributes in common, so

$$S(Q) = S(R_1) + S(R_2)$$

But what will be a good estimate for $T(Q)$?

A similar analysis as in the case of the natural gives:

$$s(R_1 \times R_2, R_1.a = R_2.d) = \frac{1}{\max(V(R_1, a), V(R_2, d))}$$

Therefore:

$$T(Q) = \left\lceil \frac{T(R_1)T(R_2)}{\max(V(R_1, a), V(R_2, d))} \right\rceil$$

If the join has many predicates, they are treated as *independent* events.

Let $Q = R_1(a, b, c) \bowtie_{a=d \wedge c=e} R_2(d, e)$. Then:

$s(R_1 \times R_2, R_1.a = R_2.d) = \frac{1}{\max(V(R_1,a),V(R_2,d))}$

$s(R_1 \times R_2, R_1.c = R_2.e) = \frac{1}{\max(V(R_1,c),V(R_2,e))}$

And:

$$T(Q) = \left\lceil \frac{T(R_1)T(R_2)}{\max(V(R_1, a), V(R_2, d)) \cdot \max(V(R_1, c)), (V(R_2, e))} \right\rceil$$

Let $Q = R_1 \bowtie R_2 \bowtie R_3$, where all three relations have attribute a in common. In this case, treat the joins $R_1 \bowtie R_2$ and $R_2 \bowtie R_3$ as independent events, arriving at:

$$T(Q) = \left\lceil \frac{T(R_1)T(R_2)T(R_3)}{\max(V(R_1, a), V(R_2, a)) \cdot \max(V(R_2, a)), (V(R_3, a))} \right\rceil$$

In general, for $Q = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_k$, we estimate $T(Q)$ by finding the selectivity of the pairwise join conditions and using their **product** for the final selectivity of the multi-way join as a whole.

Again, we do this because we assume that the events corresponding to tuples joining are independent of one another.

Other Estimates…

For joins with disjunctions $R_1 \bowtie_{c_1 \vee \ldots \vee c_n} R_2$ we can use the "complement of the complements" method to find the selectivity of the entire join predicate.

For conditions defined with inequalities $R_1 \bowtie_{a>b} R_2$, we can adapt ideas from predicate selectivity estimation for selections to find the selectivity.

The size of queries with **aggregations** is determined by the number of groupings that exist in the table.

Using $V(R, a_i)$ for each attribute a_i in the grouping set, we can estimate the total number of groups, again assuming attributes are independent of one another.

What about set operations?

Reliable estimates are possible for **set operations** that can be rewritten as conjunctions or disjunctions:

- recall $\sigma_{c_1}(R) \cup \sigma_{c_2}(R) = \sigma_{c_1 \vee c_2}(R)$ and
- $\sigma_{c_1}(R) \cap \sigma_{c_2}(R) = \sigma_{c_1 \wedge c_2}(R)$

If that's not possible, we can at least get *upper bounds* on $T(Q)$:

$$
\begin{aligned}
T(R \cup S) &= T(R) + T(S) \\
T(R \cap S) &= \min(T(R), T(S)) \\
T(R - S) &= T(R)
\end{aligned}
$$

Cost-based Query Optimization

Goal of the Query Optimizer

The goal of the query optimizer is to find **optimal query plans**. Because of **Rule #1** below, DBMSs aim for plans with the lowest possible **estimated**[8] cost.

In general, query optimization is NP-hard. In practice, the optimizer considers only a subset of the possible query plans for each query, and picks a plan with lowest estimated cost.

Rule #1 of query optimization
The time it takes the optimizer to find a good plan cannot exceed the time it would take to execute a “reasonable” plan.

[8]Recall that to find the **actual** cost, the DBMS would have to execute the plan. It makes no sense to use actual costs of multiple plans.

Query Optimization With Dynamic Programming (Sellinger, 1979)

Dynamic programming is an algorithmic problem-solving technique suitable for optimization problems where the solution builds on solutions to sub-problems.

Dynamic programming works bottom-up: decompose the problem into every atomic sub-problem and solve them; the solution with $n+1$ sub-problems builds on the lowest cost solution of size n.

$$\begin{tikzcd}
& & \pi_{a_1,\dots,a_n} & \\
& & \sigma_{c_1\wedge c_2} \arrow[u] & \\
& & \bowtie \arrow[u] & \\
& \bowtie \arrow[ur] & & T \arrow[ul] \\
R \arrow[ur] & & S \arrow[ul] &
\end{tikzcd}$$

$$\pi_{a_1,\dots,a_n} \qquad \sigma_{c_1} \qquad \sigma_{c_2} \qquad \bowtie \qquad \bowtie \qquad R \qquad S \qquad T$$

Dynamic Programming Optimizer

Step 1: for each atomic sub-problem Q, compute:

- $T(Q)$ or $B(Q)$: the estimated size of Q
- $plan(Q)$: the lowest-cost plan to compute Q
- $cost(Q)$: the actual cost to compute Q.

Step 2: build, recursively, all legal solutions of size $n+1$, considering all combinations of a solution of size n and the solution to another sub-problem. Keep the one with least cost.

Invalid partial solutions (e.g., doing a projection “before scanning a table”) are discarded.

The cost function needs to take into account **both I/O and memory** costs. However, these two costs have different units, so we cannot just add them up.

Different systems achieve this goal differently. One simple way is to consider the number of tuples that are read and computed.

For example:

- $cost(R) = T(R)$
- $cost(\rho_{S(s_1,\ldots,s_n)}(R)) = cost(R)$
- $cost(\sigma_C(R)) = T(\sigma_C(R)) + cost(R)$
- $cost(\pi_{a_1,\ldots,a_n}(R)) = T(\pi_{a_1,\ldots a_n}(R)) + cost(R)$
- $cost(R \bowtie_C S)$[9] $= T(R \bowtie_C S) + cost(R) + cost(S)$

[9] This is an in-memory nested-loop-join.

Why is Join Ordering Important?

There is a combinatorial number of plans that join N relations.

Example: $N = 4$. Here are the 5 “shapes” for a 4-way join:

$$\begin{tikzcd}
 & & & \bowtie & \\
 & & \bowtie \arrow[ur] & & U \arrow[ul] \\
 & \bowtie \arrow[ur] & & T \arrow[ul] & \\
R \arrow[ur] & & S \arrow[ul] & &
\end{tikzcd}
\quad
\begin{tikzcd}
 & \bowtie & & & \\
R \arrow[ur] & & \bowtie \arrow[ul] & & \\
 & S \arrow[ur] & & \bowtie \arrow[ul] & \\
 & & T \arrow[ur] & & U \arrow[ul]
\end{tikzcd}
\quad
\begin{tikzcd}
 & & & \bowtie & & & \\
 & \bowtie \arrow[urr] & & & & \bowtie \arrow[ull] & \\
R \arrow[ur] & & S \arrow[ul] & & T \arrow[ur] & & U \arrow[ul]
\end{tikzcd}
\quad
\begin{tikzcd}
 & \bowtie & & \\
R \arrow[ur] & & \bowtie \arrow[ul] & \\
 & \bowtie \arrow[ur] & & U \arrow[ul] \\
S \arrow[ur] & & T \arrow[ul] &
\end{tikzcd}
\quad
\begin{tikzcd}
 & & \bowtie & \\
 & \bowtie \arrow[ur] & & U \arrow[ul] \\
R \arrow[ur] & & \bowtie \arrow[ul] & \\
 & S \arrow[ur] & & T \arrow[ul]
\end{tikzcd}$$

With each shape[10] we can have any of the permutations of the relations. With relations there are $4!(\frac{6!}{(4! \cdot 3!)}) = 120$ alternatives.

With 5 relations, there are $5!(\frac{8!}{(5! \cdot 4!)}) = 1,680$ alternatives.

[10]The number of shapes comes from `https://en.wikipedia.org/wiki/Catalan_number`.

How many **actual** plans?

The space of possible plans considers, for every single one of the 120 ways of performing the 4-way join, all possible **legal rewrites** of the query (i.e., pushing selections down or not?) and, for each rewrite, all possible algorithms for each of the joins (nested-loop, hash-join, merge-join,...) and scans (table scan, index scan, if so, which one?).

Considering all these plans would almost always take longer than executing a “reasonable” one.

Left Deep Plans

Back to the $N = 4$ case.

Among all 120 plans, only $4! = 24$ are “left-deep”

$$\begin{tikzcd}
 & & & \bowtie & \\
 & & \bowtie \arrow[ur] & & U \arrow[ul] \\
 & \bowtie \arrow[ur] & & T \arrow[ul] & \\
R \arrow[ur] & & S \arrow[ul] & &
\end{tikzcd}$$

A “left-deep” binary tree is one in which all right children are leaves.

There are two main advantages of restricting the search to left-deep plans:

(1) The search space is much smaller.

(2) These plans work well in the pipeline fashion of the iterator-style query answering.

Worked example: Finding the best left-deep plan based on estimated cost.

$R(a, b, c)$ $S(c, d, e)$
$T(b, f, g)$

```
SELECT *
FROM R JOIN S
   JOIN T JOIN  U
WHERE f > 10
```

	$T(\cdot)$	$V(\cdot,\cdot)$		
R	10,000	a	b	c
		500	1000	500
S	20,000	c	d	e
		2,000	200	250
T	3,000	b	f	g
		1,500	20	2

Assume:

- $s(f > 10, T) = 0.01$
- $low(f, T) = -20$
- $high(f, T) = 20$

All plans of size 1 (i.e., with a single node) are table scans:

$$
\begin{array}{lll}
Q = R & T(Q) = 10K & cost(Q) = 10K \\
Q = S & T(Q) = 20K & cost(Q) = 20K \\
Q = T & T(Q) = 3K & cost(Q) = 3K
\end{array}
$$

The nodes $\bowtie$, $\bowtie$, and $\sigma_{f>10}$ cannot be used alone and thus cannot be part of a single-node plan

There is **only one** valid plan of size 2:

$$Q = \begin{array}{c} \sigma_{f>10} \\ \uparrow \\ T \end{array} \qquad T(Q) = 3K \cdot 0.01 = 30 \quad cost(Q) = 3.03K$$

NOTE: $V(\sigma_{f>10}(T), b) = 30$, although the original table had a lot more distinct values of b (recall slide 101).

Now, plans with a **single join**, starting with those of size 3:

$$Q = \begin{tikzcd} & \bowtie & \\ S \arrow[ur] & & R \arrow[ul] \end{tikzcd} \qquad T(Q) = \frac{10K \cdot 20K}{\max(500, 2K)} = 100K \qquad cost(Q) = 130K$$

$$Q = \begin{tikzcd} & \bowtie & \\ R \arrow[ur] & & T \arrow[ul] \end{tikzcd} \qquad T(Q) = \frac{10K \cdot 3K}{\max(1K, 1.5K)} = 20K \qquad cost(Q) = 33K$$

$$Q = \begin{tikzcd} & \bowtie & \\ S \arrow[ur] & & T \arrow[ul] \end{tikzcd} \qquad T(Q) = 20K \cdot 3K = 60M \qquad cost(Q) = 60.023M$$

NOTE: query plan $\begin{tikzcd} & \bowtie & \\ S \arrow[ur] & & T \arrow[ul] \end{tikzcd}$ has a high cost because it is a *cross product*: there are no attributes common to S and T.

- A modern optimizer will immediately prune that sub-plan as it cannot be better than any other plan.

Now, plans of size 4 (single-join with T and $\sigma_{f>10}$).

We will ignore the plans involving the cross product $S \bowtie T$, and consider the only two plans involving R and T:

$$Q = \begin{tikzcd} & \bowtie & \\ R \arrow[ur] & & \sigma_{f>10} \arrow[ul] \\ & & T \arrow[u] \end{tikzcd}$$

$$T(Q) = \frac{10K \cdot 30}{\max(1K, 30)} = 300$$

$$cost(Q) = 300 + 10K + 3.03K = 13.33K$$

For the next one, we re-use the work done for plans of size 3.

$$Q = \begin{tikzcd} & \sigma_{f>10} & \\ & \bowtie \arrow[u] & \\ R \arrow[ur] & & T \arrow[ul] \end{tikzcd}$$

$$T(Q) = 20K \cdot 0.01 = 200$$

$$cost(Q) = 200 + 33K = 33.2K$$

Now for plans of size 5, which would be the best plans of size 3 plus another join (and another table).

$$Q = \begin{tikzcd}[column sep=small, row sep=small] & & \bowtie & \\ & \bowtie \arrow[ur] & & T \arrow[ul] \\ S \arrow[ur] & & R \arrow[ul] & \end{tikzcd}$$

$$\begin{aligned} T(Q) &= \frac{T(R, S, \bowtie)T(T)}{\max(V(b, R \bowtie S), V(b, T))} \\ &= \frac{100K \cdot 3K}{\max(1K, 1.5K)} = 200K \\ cost(Q) &= 200K + 130K + 3K = 363K \end{aligned}$$

$$Q = \begin{tikzcd}[column sep=small, row sep=small] & & \bowtie & \\ & \bowtie \arrow[ur] & & S \arrow[ul] \\ R \arrow[ur] & & T \arrow[ul] & \end{tikzcd}$$

$$\begin{aligned} T(Q) &= \frac{20K \cdot 20K}{\max(500, 2K)} = 200K \\ cost(Q) &= 200K + 33K + 20K = 253K \end{aligned}$$

Again, we ignore the plan with $S \bowtie T$.

An now for plans of size 6, that will include all nodes.

Option 1: add $\sigma_{f>10}$ to the **best** plan of size 5.

$$\begin{tikzcd}
 & & & \sigma_{f>10} \\
 & & & \bowtie \arrow[u] \\
 & & \bowtie \arrow[ur] & & S \arrow[ul] \\
Q = & R \arrow[ur] & & T \arrow[ul]
\end{tikzcd}$$

$$T(Q) = 200K \cdot 0.01 = 20K$$

$$costQ = 20K + 253K = 273K$$

Option 2: join S to the **best** plan of size 4.

$$\begin{tikzcd}
 & & & \bowtie \\
 & & \bowtie \arrow[ur] & & S \arrow[ul] \\
 & R \arrow[ur] & & \sigma_{f>10} \arrow[ul] \\
Q = & & & T \arrow[u]
\end{tikzcd}$$

$$T(Q) = \frac{300 \cdot 20K}{\max(500, 2K)} = 3K$$

$$costQ = 3K + 13.33K + 20K = 36.33K$$

And we are done!

Notes

The Sellinger optimizer can handle all operators in a SQL query, choosing the best place for each operator.

(1) Note that it “figured out” that pushing selections down the tree is the right thing to do. Although we would save more time by always pruning plans with “high” selections.

(2) However, even with aggressive pruning, Dynamic programming is too expensive to be practical.

  - Commercial optimizers often use many heuristics and avoid enumerating all plans as we did here.

(3) Machine learning can help by providing better cost estimates and also by helping solve the optimization problem.

Query Plans with Indexes

Recall that indexes as *associative access methods* that allow the DBMS to quickly find tuples based on the values of their attributes.

B+ tree indexes, hash tables and hash sets can be used to implement many query plan operators: selections, joins (and their variants), all set operators, duplicate elimination, and grouping.

The next slides give some examples of the use of indexes.

We start by looking at **base indexes**, or indexes created by the database administrator, and discuss briefly the idea of on-the-fly indexing, which is to create an index with the result of a sub-query to be used once.

The table below indicates, generally, which kinds of predicates can be improved by the use of **indexes**.

	$a_i = v$	$a_i \geq v$	a_i `IN ()`	$a_i \neq v$	a_i `NOT IN ()`
B+ tree	✓	✓[†]	✓	✗[‡]	✓
Hash table	✓	✗	✓	✗[‡]	✓

[†] If the B+tree is a secondary index and the selectivity of the predicate is high, a table scan might be better.

[‡] These predicates often have very high selectivity, so the table scan is almost always better.

How are Indexes Chosen?

A predicate a_i <op> v in the **WHERE** clause can be answered using an index only if a_i is a prefix of the index key.

Example:

```
SELECT director
FROM Movie
WHERE title='Ghostbusters' AND year=1984
```

Useful Indexes

`Movie(title)`, `Movie(year)`,

`Movie(title,imdb)`, …

Not Useful

`Movie(imdb,title)`,

`Movie(director)` …

If multiple useful indexes exist, the DBMS picks the one with lowest estimated cost, which depends on the selectivity of the predicate!

$$
\begin{array}{r|l}
T(\text{Movie}) = 100,000 & V(\text{Movie}, \text{title}) = 95,000 \\
S(\text{Movie}) = 92 & V(\text{Movie}, \text{year}) = 32 \\
B(\text{Movie}) = 2,000 & V(\text{Movie}, \text{imdb}) = 50 \\
 & V(\text{Movie}, \text{director}) = 25,000
\end{array}
$$

Suppose we have the following B+ tree indexes with 100 pointers/node.

```
> CREATE INDEX IDX1 ON Movie(Title);
> CREATE INDEX IDX2 ON Movie(Title,Year);
> CREATE INDEX IDX3 ON Movie(Year);
```

In all cases, 3 levels are enough to index all keys.

How many blocks (at the leaf level) of each index will have tuples matching the query?

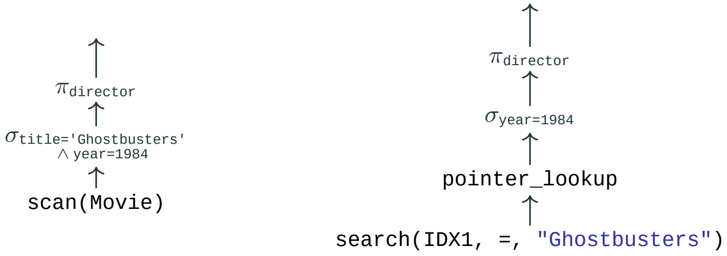
$$T(\sigma_{\texttt{title='Ghostbusters'}}(\texttt{Movie})) = \left\lceil \frac{100,000}{95,000} \right\rceil = 2$$

$$T(\sigma_{\texttt{year=1984}}(\texttt{Movie})) = \left\lceil \frac{100,000}{32} \right\rceil = 3,125$$

We can thus expect to find:

- all movies with that title in a single **leaf** block of `IDX1`
- all movies from 1984 in 32 **leaf** blocks of `IDX3`
- all movies with that title/year in a single **leaf** block of `IDX2`

```
SELECT director
FROM Movie
WHERE title='Ghostbusters' AND year=1984
```

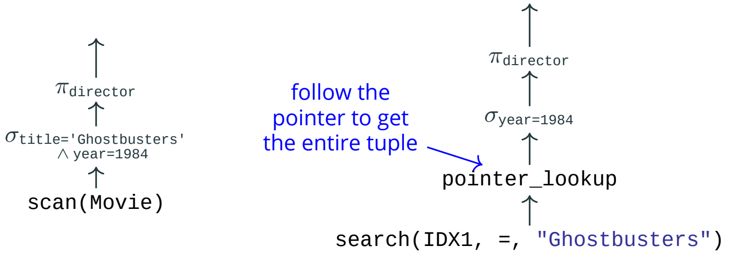


π_{director}
$\sigma_{\text{title='Ghostbusters'} \wedge \text{year=1984}}$
scan(Movie)
π_{director}
$\sigma_{\text{year=1984}}$
pointer_lookup
search(IDX1, =, "Ghostbusters")

Estimated I/O costs:

- left plan: 2,000 blocks
- right plan: 3 + 2 = 5 blocks[11]

```
SELECT director
FROM Movie
WHERE title='Ghostbusters' AND year=1984
```


$\pi_{\texttt{director}}$
$\sigma_{\texttt{title='Ghostbusters'} \wedge \texttt{year=1984}}$
scan(Movie)
follow the pointer to get the entire tuple
$\pi_{\texttt{director}}$
$\sigma_{\texttt{year=1984}}$
pointer_lookup
search(IDX1, =, "Ghostbusters")

Estimated I/O costs:

- left plan: 2,000 blocks
- right plan: 3 + 2 = 5 blocks[11]

[11] 3 I/Os on the index (root to leaf path) + 2 lookups on the table.

Cost recap

The cost of retrieving tuples from a table R satisfying a predicate p using a a B+ Tree with $|I|$ blocks at the *leaf level* is the sum of:

1. Number of inner blocks in the path from root to the first index leaf (usually 2 or 3).
2. Number of blocks of the index that will be read: $\lceil s(p) \cdot |I| \rceil$.
3. Number of pointer lookups: $T(\sigma_p(R))$ (unless this is a primary index and the table is sorted as the index).

With a hash table, the estimated cost is $T(\sigma_p(R)) \cdot 2$: one read in the hash table and another for the pointer traversal to the table.

Covering Index

The `pointer_lookup` operator pulls a pointer to a tuple in `Movie` returned by the index search operator and goes to the block of file of the `Movie` table that contains the respective tuple and fetches it.

This operator is needed because the query uses attributes of `Movie` that are not present in the index record (`year` and `director`).

An index with all attributes from the table used in a query is called a covering index.

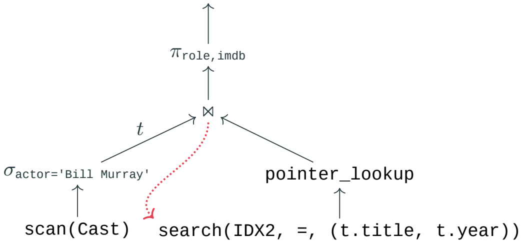
```
SELECT year
FROM Movie
WHERE title='Ghostbusters'
```

`IDX2` is a covering index for the query above.

Index-Based Joins

Depending on whether the index is primary or not, and on the selectivity of a join predicate, the DBMS might use a base index to speed up a nested-loop join:

```
SELECT role, imdb
FROM Movie
  JOIN Cast
WHERE actor='Bill Murray';
```



$\pi_{\texttt{role,imdb}}$
$\bowtie$
t
$\sigma_{\texttt{actor='Bill Murray'}}$
pointer_lookup
scan(Cast)
search(IDX2, =, (t.title, t.year))

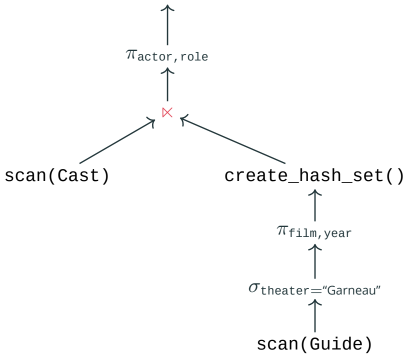
The I/O cost is bound by $s(\text{Cast, actor='Bill Murray'})$.

Of course, if there was an index on `Cast(actor)`, it could also be used instead of the table scan in the example above.

On the Fly Indexing

The DBMS might create an index/hash table/hash set with the results of a sub-query to speed up a join.

```
SELECT actor, role
FROM Cast
WHERE (title,year) IN (
    SELECT film, year
    FROM Guide
    WHERE theater='Garneau');
```



$\pi_{\texttt{actor,role}}$
$\ltimes$
scan(Cast)
create_hash_set()
$\pi_{\texttt{film,year}}$
$\sigma_{\texttt{theater}=\text{“Garneau”}}$
scan(Guide)

Recall the **semijoin** $R \ltimes S$ returns tuples from R that match a tuple in S with respect to the join attributes (`title,year`).

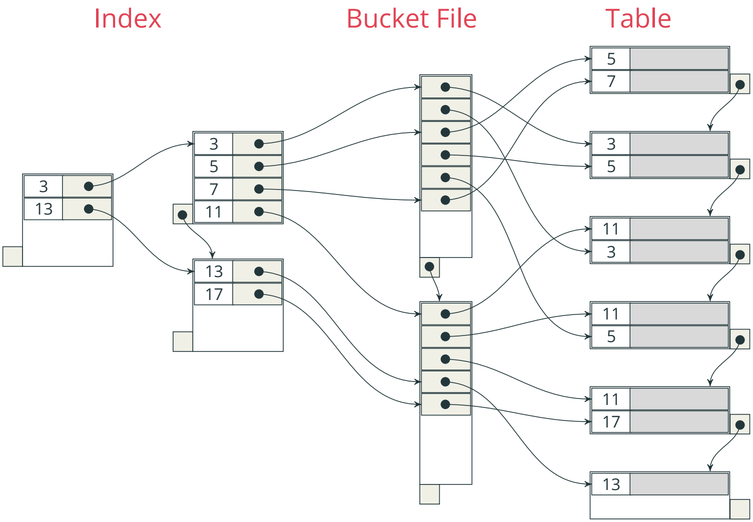
Answering Complex Predicates with Secondary Indexes

Primary indexes allow the fastest access to tuples in the corresponding table because the tuples in the table file are sorted in the same way as the keys in the index.
- these indexes are sometimes called **clustered indexes**

The problem is there can be only one primary index per table.

A way to bring some of the benefits of primary to secondary indexes is to **cluster tuple identifiers** based on search key in an a “bucket file” and have a primary index over that file.

Secondary Indexes with Bucket Files



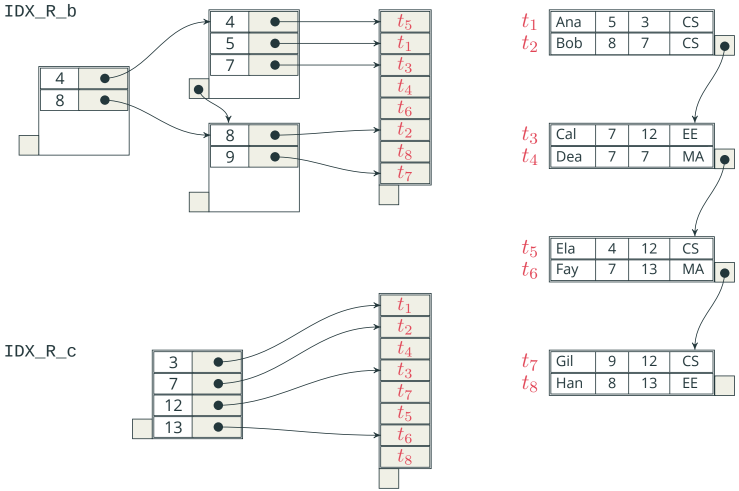
Finding Tuples

Note that the index has no duplicate entries for tuples with the same value.

To find all tuples within a range (v_1, v_2), we need two pointers.

```
**answer** $\leftarrow \langle\rangle$
$p_1 \leftarrow$ first tuple whose key is at least $v_1$
$p_2 \leftarrow$ first tuple whose key is greater than $v_2$ (or <EOF>)
open bucket file at position $p_1$;
**while** $p_1$ is strictly before $p_2$ **do**
    append tuple id in $p_1$ to **answer**;
    advance $p_1$
**return answer**
```

Example: table $R(a, b, c, d)$;
primary index on $R.a$;
secondary indexes on $R.b$ and $R.c$.

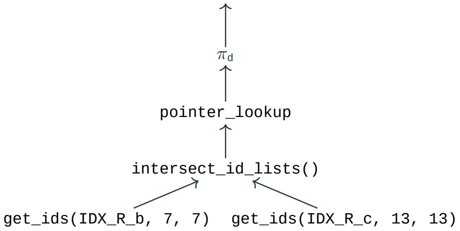


t_1	Ana	5	3	CS
t_2	Bob	8	7	CS
t_3	Cal	7	12	EE
t_4	Dea	7	7	MA
t_5	Ela	4	12	CS
t_6	Fay	7	13	MA
t_7	Gil	9	12	CS
t_8	Han	8	13	EE

Complex selections with tuple ids

The DBMS can solve an arbitrary selection using lists of tuple ids that satisfy each predicate separately.

```
SELECT d
FROM R
WHERE b = 7 AND c = 13;
```



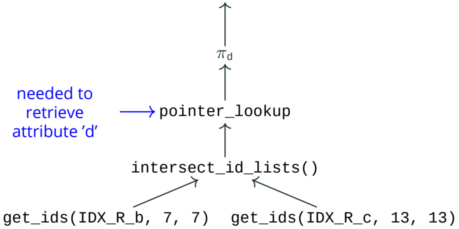
`get_ids(IDX_R_b, b=7)` = $\langle t_3, t_4, t_6 \rangle$

`get_ids(IDX_R_c, c=13)` = $\langle t_6, t_8 \rangle$

Complex selections with tuple ids

The DBMS can solve an arbitrary selection using lists of tuple ids that satisfy each predicate separately.

```
SELECT d
FROM R
WHERE b = 7 AND c = 13;
```



`get_ids(IDX_R_b, b=7)` = $\langle t_3, t_4, t_6 \rangle$

`get_ids(IDX_R_c, c=13)` = $\langle t_6, t_8 \rangle$

Conjunction $\sigma_{c_1 \wedge c_2}(R)$

$$\begin{tikzcd}
 & \text{list intersection} & \\
 & \texttt{intersect\_id\_lists} \arrow[u] & \\
\texttt{get\_ids(IDX\_R,}c_1\texttt{)} \arrow[ur] & & \texttt{get\_ids(IDX\_R,}c_2\texttt{)} \arrow[ul]
\end{tikzcd}$$

Disjunction $\sigma_{c_1 \vee c_2}(R)$

$$\begin{tikzcd}
 & \text{list union} & \\
 & \texttt{union\_id\_lists} \arrow[u] & \\
\texttt{get\_ids(IDX\_R,}c_1\texttt{)} \arrow[ur] & & \texttt{get\_ids(IDX\_R,}c_2\texttt{)} \arrow[ul]
\end{tikzcd}$$

Because tuple ids are **clustered** in the bucket files, this approach can lead to very fast answers!

I/O cost

How many blocks are read to find (the identifiers of) all tuples satisfying $v_1 \leq a_i \leq v_2$?

(1) number of blocks of the index to find the pointers: $O(h)$, where h is the height of the B+tree

(2) number of blocks of the bucket file: $\lceil s(R, v_1 \leq a_i \leq v_2) \cdot (\text{# blocks in bucket file}) \rceil$

The query plan incurs more I/O operations to bring the pointers from the base table, so that the attributes can be read.

Example: $R(a, b)$; 1 million tuples:

- 1000 distinct values of attribute a;
- 200 distinct values of attribute b

Each disk block can hold:

- 50 tuples or
- 200 key-pointer pairs or
- 400 tuple ids

file sizes:

- data: 20,000 blocks
- B+ tree on $R.a$:

Example: $R(a,b)$; 1 million tuples:

- 1000 distinct values of attribute a;
- 200 distinct values of attribute b

Each disk block can hold:

- 50 tuples or
- 200 key-pointer pairs or
- 400 tuple ids

file sizes:

- data: 20,000 blocks
- B+ tree on $R.a$: 2 levels
- bucket file $R.a$:

Example: $R(a, b)$; 1 million tuples:

- 1000 distinct values of attribute a;
- 200 distinct values of attribute b

Each disk block can hold:

- 50 tuples or
- 200 key-pointer pairs or
- 400 tuple ids

file sizes:

- data: 20,000 blocks
- B+ tree on $R.a$: 2 levels
- bucket file $R.a$: 2,500 blocks
- B+ tree on $R.b$:

Example: $R(a, b)$; 1 million tuples:

- 1000 distinct values of attribute a;
- 200 distinct values of attribute b

Each disk block can hold:

- 50 tuples or
- 200 key-pointer pairs or
- 400 tuple ids

file sizes:

- data: 20,000 blocks
- B+ tree on $R.a$: 2 levels
- bucket file $R.a$: 2,500 blocks
- B+ tree on $R.b$: 1 level
- bucket file $R.b$:

Example: $R(a, b)$; 1 million tuples:

- 1000 distinct values of attribute a;
- 200 distinct values of attribute b

Each disk block can hold:

- 50 tuples or
- 200 key-pointer pairs or
- 400 tuple ids

file sizes:

- data: 20,000 blocks
- B+ tree on $R.a$: 2 levels
- bucket file $R.a$: 2,500 blocks
- B+ tree on $R.b$: 1 level
- bucket file $R.b$: 2,500 blocks

I/O cost of $\sigma_{a=k_1 \wedge b=k_2}(R)$:

Option 1 – table scan: $20,000$

Option 2 – merging lists:

Example: $R(a, b)$; 1 million tuples:

- 1000 distinct values of attribute a;
- 200 distinct values of attribute b

Each disk block can hold:

- 50 tuples or
- 200 key-pointer pairs or
- 400 tuple ids

file sizes:

- data: 20,000 blocks
- B+ tree on $R.a$: 2 levels
- bucket file $R.a$: 2,500 blocks
- B+ tree on $R.b$: 1 level
- bucket file $R.b$: 2,500 blocks

I/O cost of $\sigma_{a=k_1 \wedge b=k_2}(R)$:

Option 1 – table scan: $20,000$

Option 2 – merging lists:

$3 +$

Example: $R(a, b)$; 1 million tuples:
- 1000 distinct values of attribute a;
- 200 distinct values of attribute b

Each disk block can hold:
- 50 tuples or
- 200 key-pointer pairs or
- 400 tuple ids

file sizes:

- data: 20,000 blocks
- B+ tree on $R.a$: 2 levels
- bucket file $R.a$: 2,500 blocks
- B+ tree on $R.b$: 1 level
- bucket file $R.b$: 2,500 blocks

I/O cost of $\sigma_{a=k_1 \wedge b=k_2}(R)$:

Option 1 – table scan: $20,000$

Option 2 – merging lists:

$3 + \lceil 2,500 \cdot s(R, \text{a} = k_1) +$

Example: $R(a, b)$; 1 million tuples:

- 1000 distinct values of attribute a;
- 200 distinct values of attribute b

Each disk block can hold:

- 50 tuples or
- 200 key-pointer pairs or
- 400 tuple ids

file sizes:

- data: 20,000 blocks
- B+ tree on $R.a$: 2 levels
- bucket file $R.a$: 2,500 blocks
- B+ tree on $R.b$: 1 level
- bucket file $R.b$: 2,500 blocks

I/O cost of $\sigma_{a=k_1 \wedge b=k_2}(R)$:

Option 1 – table scan: $20,000$

Option 2 – merging lists:

$3 + \lceil 2,500 \cdot s(R, \texttt{a} = k_1) + 2,500 \cdot s(R, \texttt{b} = k_2) \rceil +$

Example: $R(a, b)$; 1 million tuples:
- 1000 distinct values of attribute a;
- 200 distinct values of attribute b

Each disk block can hold:
- 50 tuples or
- 200 key-pointer pairs or
- 400 tuple ids

file sizes:

- data: 20,000 blocks
- B+ tree on $R.a$: 2 levels
- bucket file $R.a$: 2,500 blocks
- B+ tree on $R.b$: 1 level
- bucket file $R.b$: 2,500 blocks

I/O cost of $\sigma_{a=k_1 \wedge b=k_2}(R)$:

Option 1 – table scan: $20,000$

Option 2 – merging lists:

$$3 + \lceil 2,500 \cdot s(R, \mathsf{a} = k_1) + 2,500 \cdot s(R, \mathsf{b} = k_2) \rceil + \min\left(s(R, \mathsf{a} = k_1), s(R, \mathsf{b} = k_2)\right) \cdot T(R)$$

I/O cost of $\sigma_{a=k_1 \wedge b=k_2}(R)$:

Option 3 – Scan a single index:

If one predicate has a *much lower* selectivity than the other, the DBMS can read from one of the indexes and evaluate the other predicate after the tuple is retrieved from the database.

$$\begin{tikzcd} {} \\ \sigma_{\texttt{b}=k_2} \arrow[u] \\ \texttt{pointer\_lookup} \arrow[u] \\ \texttt{get\_ids(IDX\_on\_a, a=}k_1\texttt{)} \arrow[u] \end{tikzcd}$$

Ex: Assuming $s(R, \texttt{a} = k_1) \ll s(R, \texttt{b} = k_2)$

The cost would be

$$(2 +$$

I/O cost of $\sigma_{a=k_1 \wedge b=k_2}(R)$:

Option 3 – Scan a single index:

If one predicate has a *much lower* selectivity than the other, the DBMS can read from one of the indexes and evaluate the other predicate after the tuple is retrieved from the database.

$$\begin{tikzcd} {} \\ \sigma_{\texttt{b}=k_2} \arrow[u] \\ \texttt{pointer\_lookup} \arrow[u] \\ \texttt{get\_ids(IDX\_on\_a, a=}k_1\texttt{)} \arrow[u] \end{tikzcd}$$

Ex: Assuming $s(R, \texttt{a} = k_1) \ll s(R, \texttt{b} = k_2)$

The cost would be

$$(2 + \lceil 2,500 \cdot s(R, \texttt{a} = k_1) \rceil) +$$

I/O cost of $\sigma_{a=k_1 \wedge b=k_2}(R)$:

Option 3 – Scan a single index:

If one predicate has a *much lower* selectivity than the other, the DBMS can read from one of the indexes and evaluate the other predicate after the tuple is retrieved from the database.

$$\begin{tikzcd}[row sep=small]
{} \\
\sigma_{\texttt{b}=k_2} \arrow[u] \\
\texttt{pointer\_lookup} \arrow[u] \\
\texttt{get\_ids(IDX\_on\_a, a=}k_1\texttt{)} \arrow[u]
\end{tikzcd}$$

Ex: Assuming $s(R, \texttt{a} = k_1) \ll s(R, \texttt{b} = k_2)$

The cost would be

$$(2 + \lceil 2,500 \cdot s(R, \texttt{a} = k_1 \rceil) + \ ) \ + \lceil T(R) \cdot s(R, \texttt{a} = k_1) \rceil$$

Other SQL Constructs

Aggregation

Aggregation is executed by first finding groupings of tuples where they all agree on some attributes, then applying set functions to each group:

```
SELECT a_1, ..., a_k, fn_1(), ..., fn_i()
FROM T_1 [,  ... [, T_m]]
WHERE c_1 [ AND  ... [AND c_j]]
GROUP BY a_1,  ...,  a_k
HAVING hc_1 [ AND ... [ AND hc_l ]]
ORDER BY col_no [ASC | DESC]
```

$$\begin{tikzcd}
\texttt{sort\_by(col\_no)} \\
\pi_{\texttt{a}_1,\ldots,\texttt{a}_k,\texttt{fn}_1(),\ldots,\texttt{fn}_i()} \arrow[u] \\
\sigma_{\texttt{hc}_1 \wedge \cdots \wedge \texttt{hc}_l} \arrow[u] \\
\gamma_{\texttt{a}_1,\ldots,\texttt{a}_k,\texttt{fn}_1(),\ldots,\texttt{fn}_i()} \arrow[u] \\
\sigma_{\texttt{c}_1 \wedge \cdots \wedge \texttt{c}_j} \arrow[u] \\
T_1 \times \cdots \times T_m \arrow[u]
\end{tikzcd}$$

Computing $\gamma_{\texttt{a}_1,\ldots,\texttt{a}_k,\texttt{fn}_1(),\ldots,\texttt{fn}_m()}$ with a **hash map**:

(1) Hash every tuple resulting from $\sigma_{\texttt{c}_1 \wedge \cdots \wedge \texttt{c}_j}$ using $\texttt{a}_1, \ldots, \texttt{a}_k$ as the hash key;

(2) Go through each bucket separately, accounting for *collisions*;

  (a) Compute each of $\texttt{fn}_1(), \ldots, \texttt{fn}_m()$ for each group;

  (b) Add tuple $\texttt{a}_1, \ldots, \texttt{a}_k, \texttt{fn}_1(), \ldots, \texttt{fn}_m()$ to the output.

Computing $\gamma_{\texttt{a}_1,\ldots,\texttt{a}_k,\texttt{fn}_1(),\ldots,\texttt{fn}_m()}$ with **sorting**:

(1) Insert tuples resulting from $\sigma_{\texttt{c}_1\wedge\cdots\wedge\texttt{c}_j}$ into buffers in memory (or disk if there is no memory available);

(2) Sort the tuples on $\texttt{a}_1, \ldots, \texttt{a}_k$;

(3) Scan the sorted tuples in order, grouping together those with the same values for $\texttt{a}_1, \ldots, \texttt{a}_k$;

   (a) Compute each of $\texttt{fn}_1(), \ldots, \texttt{fn}_m()$ for each group;

   (b) Add tuple $\texttt{a}_1, \ldots, \texttt{a}_k, \texttt{fn}_1(), \ldots, \texttt{fn}_m()$ to the output.

An alternative would be to store the tuples from $\sigma_{\texttt{c}_1\wedge\cdots\wedge\texttt{c}_j}$ into an index in memory using $\texttt{a}_1, \ldots, \texttt{a}_k$ as indexing key.

Blocking Operators

Recall that with the iterator model tuples of the answer are sent to the preceding operator in the query plan *one by one*.

Note that the aggregation operator, in general, requires one to collect **all tuples** from the incoming operator (to form the groups) before any output tuple can be produced[12].

An operator that requires materializing intermediate results is called a BLOCKING operator.

Blocking operators are bad for performance–this is why some systems (especially noSQL ones) do not support aggregation at all.

[12]An exception to that rule would be if the incoming tuples were already sorted by the group by attributes, which is unlikely to happen too often.

Nested Queries

Besides table expressions SQL also allows full sub-queries or as operands in set membership testing (**IN**/**NOT IN** expressions), used typically in the **WHERE** clause.

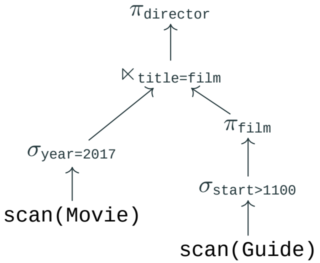
```
SELECT $a_1, a_2, \ldots, a_n$
FROM $T_1$ [, $T_2$ [, ... [, $T_m$]]]
WHERE $c_1$ AND … AND $c_k$ AND|OR $a_i$ [NOT] IN (sub-query)
```

The nested `sub-query` can be *correlated* with the main (outer) query if they share some tuple variable.

It should be easy to see that set memberships can be rewritten as semijoins.

Semijoins with *non-correlated* nested queries can be done efficiently with hashing, sorting or indexing (using lookups).

```
SELECT director
FROM Movie
WHERE year=2017 AND title
      IN (SELECT film
         FROM Guide
         WHERE start>1100);
```

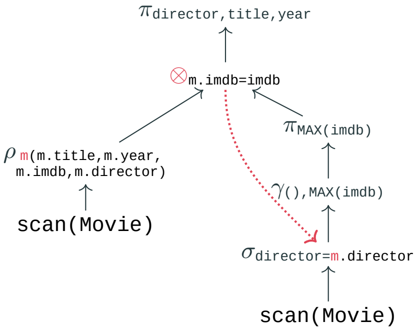


π_{director}
$\ltimes_{\text{title=film}}$
$\sigma_{\text{year=2017}}$
π_{film}
$\sigma_{\text{start>1100}}$
scan(Movie)
scan(Guide)

Note that when evaluating an **IN** predicate we do not need to consider *all possible* matches. Instead, we can send tuple from the LHS as soon as a match is found.

At first it may seem that computing set membership with *correlated* nested queries requires computing the sub-query for each tuple of the outer query, using some *parameterized* iterator for the semijoin:

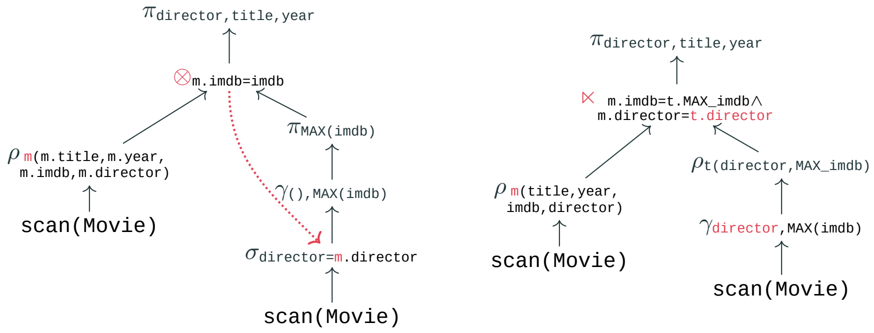
```
SELECT m.director, m.title, m.year
FROM Movie m
WHERE m.imdb
      IN (SELECT MAX(imdb)
         FROM Movie
         WHERE director=m.director);
```



$\pi_{\texttt{director,title,year}}$
$\otimes_{\texttt{m.imdb=imdb}}$
$\rho_{\texttt{m(m.title,m.year,m.imdb,m.director)}}$
scan(Movie)
$\pi_{\texttt{MAX(imdb)}}$
$\gamma_{\texttt{(),MAX(imdb)}}$
$\sigma_{\texttt{director=m.director}}$
scan(Movie)

A parameterized semijoin operator might be very efficient in a query like this, especially if there is an index on `director` *and* the join selectivity is low.

An alternative is to rewrite the query, bringing the selection “up” to the semijoin:



Queries can also be de-correlated at the syntactic (SQL) level by the pre-processor. Then the query can be optimized like any other:

```
SELECT m.director, m.title, m.year
FROM Movie m
WHERE m.imdb
      IN (SELECT MAX(imdb)
         FROM Movie
         WHERE director=m.director);
```

```
WITH maxIMDB (director,imdb) AS(
  SELECT director, MAX(imdb)
  FROM Movie
  GROUP BY director
)
SELECT m.director, m.title, m.year
FROM Movie m, maxIMDB mi
WHERE m.director = mi.director
      AND m.imdb = mi.imdb
```

What about `NOT IN`?

It should be easy to see that `NOT IN` can be easily handled within the iterator for the semijoin operator, by reversing the test and returning a tuple from the outer table that *does not* match any tuple in the inner table.

$$\begin{tikzcd}
 & {} & \\
 & \ltimes \arrow[u] & \\
exp_R \arrow[ur] & & exp_S \arrow[ul]
\end{tikzcd}$$

Sub-queries within EXISTS

Correlated sub-queries inside **EXISTS** can be rewritten using [**NOT**] **IN**, and evaluated using semijoins:

```
SELECT m.director, m.title, m.year
FROM Movie m
WHERE EXISTS (
   SELECT *
   FROM Guide, Cinema
   WHERE film=m.title AND
         year=m.year AND
         name=theater AND
         address LIKE "%Edmonton%"
);
```

```
SELECT director, title, year
FROM Movie
WHERE title || year IN (
   SELECT film || year
   FROM Guide, Cinema
   WHERE name=theater AND
         address LIKE "%Edmonton%"
);
```

NOTE: in SQL, the **IN** operator is defined for a single value expression on the LHS; thus, the example above uses string concatenation to simulate checking set membership of tuples. A real semijoin operator would not have that limitation of course.

Heuristic Optimizers

Cost-based query optimizers need to consider a large number of alternative plans to find the one with the least estimated costs.

(1) The more algorithms and kinds of indexes the DBMS has, the bigger the search space becomes!
(2) The optimizer cannot take longer than what a “good enough” plan would!

Therefore, many DBMSs use both query rewriting strategies and heuristic “rules-of-thumb” instead of considering all possible alternatives.

We have seen many heuristic optimizations already, and we will look at a few more next.

Query Rewriting

Goal: either change the SQL query itself or the query plan (using algebraic equivalence rules from the algebra) in order to simplify the execution of the query.

Some algebraic rewrite rules are guaranteed to never make the plan worse and are always applied.

(1) “Push down” selections.

(2) Rewrite intersections and unions as queries with conjunctions or disjunctions of predicates.

Query Rewriting: De-Correlating Nested Queries

A correlated nested query needs to be evaluated once for every tuple in the outer query, which can be very slow.

But it is sometimes possible to materialize the result of the entire nested query once, and then use a regular join/semi-join algorithm to compute the final answer.

```sql
SELECT m.director, m.title, m.year
FROM Movie m
WHERE m.imdb
      IN (SELECT MAX(imdb)
         FROM Movie
         WHERE director=m.director);
```

```sql
SELECT m.director, m.title, m.year
FROM Movie m
WHERE m.director || m.imdb
      IN (SELECT director || MAX(imdb)
         FROM Movie
         GROUP BY director);
```

Query Rewrite: Flattening Nested Queries

Instead of having the optimizer deal with every nuance of SQL, it is best to simplify complex SQL and **re-use** optimization code for simpler queries.

The `IN` operator (a semi-join) can be rewritten as a regular join:

```
SELECT director
FROM Movie
WHERE year=2019 AND title
      IN (SELECT film
         FROM Guide
         WHERE start>1100);
```

```
SELECT m.director
FROM Movie m, Guide g
WHERE m.year=2019
   AND m.year=g.year
   AND m.title=g.film
   AND g.start>1100;
```

Query Rewrite: Restricting Nested Queries

THe DBMS can exploit primary/foreign key dependencies between tables to further restrict nested queries, reducing the number of tuples that need to be joined.

```
SELECT director
FROM Movie
WHERE year=2019 AND title
      IN (SELECT film
         FROM Guide
         WHERE start>1100);
```

```
SELECT director
FROM Movie
WHERE year=2019 AND title
      IN (SELECT film
         FROM Guide
         WHERE start>1100
           AND year=2019);
```

“Rule-of-Thumb” Optimizations

DBMSs will attempt to reduce the time to execute a query in every way possible. When choosing a DBMS, you should look the space of optimizations and improvements that they do.

For instance, instead of evaluating all atomic predicates in the `WHERE` clause, the DBMS can simplify the logical expression so that it can stop evaluating atoms when each clause is satisfied or falsified.

```
WHERE ... m.director='Kubrick' AND c.actor='Tom Cruise' ...
      OR NOT (m.year=2014 AND m.imdb<7)
```

Can be rewritten as:

```
m.year <> 2014 OR m.imdb >= 7 OR
    (m.director='Kubrick' AND c.actor='Tom Cruise')!
```

Join Ordering

The dynamic-programming join ordering algorithm we considered finds the optimal ordering and thus requires exponential time in the worst case.

Most DBMSs use reasonable approximation algorithms that run in polynomial time. Although they will not give the best ordering in many cases, they will still give good orderings most of the time.

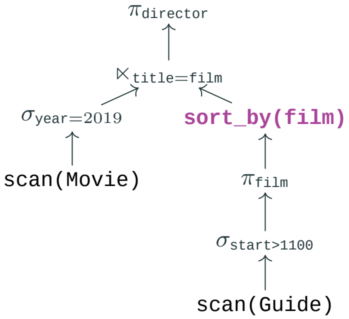
SQLite

SQLite uses a polynomial time algorithm for join ordering and uses the number of indexes on a table as its cost.

Sort Intermediate Relations

If no suitable indexes exist to help with a query, a good strategy is to sort the results of the sub-expressions and answer the join using merge-sort in memory.

```
SELECT director
FROM Movie
WHERE year=2019 AND title
      IN (SELECT film
         FROM Guide
         WHERE start>1100);
```



$\pi_{\texttt{director}}$
$\ltimes_{\texttt{title}=\texttt{film}}$
$\sigma_{\texttt{year}=2019}$
sort_by(film)
scan(Movie)
$\pi_{\texttt{film}}$
$\sigma_{\texttt{start>1100}}$
scan(Guide)