Summary

Smonman
April 7, 2024

Contents

1 Storage

1.1 Disk Space Manager

- communicates with the I/O controller
- initiates I/O
- issues *allocate*, *deallocate*, *read* and *write* commands
- abstracts from the details of the underlaying storage:
  - provides the concept of a *page* (typically 4 kB to 64 kB) as a unit of storage to the remaining system
  - maintains mapping *page-number* $\mapsto$ *physical-location*
  - higher layers of the DBMS see no hardware details, only a collection of pages

1.1.1 Implementation

- either uses file system functions of the OS, or
- implements its own disk management (*raw disk access*)

1.2 Buffer Manager

- moves data between disk and main memory
- manages a designated main memory area (*buffer pool*)
- loads disk pages into the buffer pool and writes them back to disk if they have been modified
- stores additional information:
  - is the page used?
  - has the contents of the page been modified?

1.3 Files and Access Methods Layer

1.3.1 Structure

The database primarily manages tables of tuples (*records*) and indices. All records of a table form a *file*.

The Files and Access Methods Layer manages:

- the pages of a file
- the records within a page

The access to these records by other software layers is managed via *RIDs* (records IDs). Sometimes they are also called *TIDs* (tuple IDs).

These IDs work like *pointers* to the record. Usually a RID consists of a page number and an offset into this page.

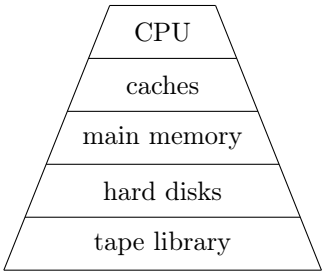
1.4 Storage Hierarchy

There are three different types of storage:

- *Primary Storage*: transient (content is lost, when the DBMS is shut down)
  - CPU registers
  - caches
  - main memory
- *Secondary Storage*: persistent (content is not lost)
  - magnetic disks
  - solid-state drives
- *Tertiary Storage*: removable (used for achives)
  - Tape

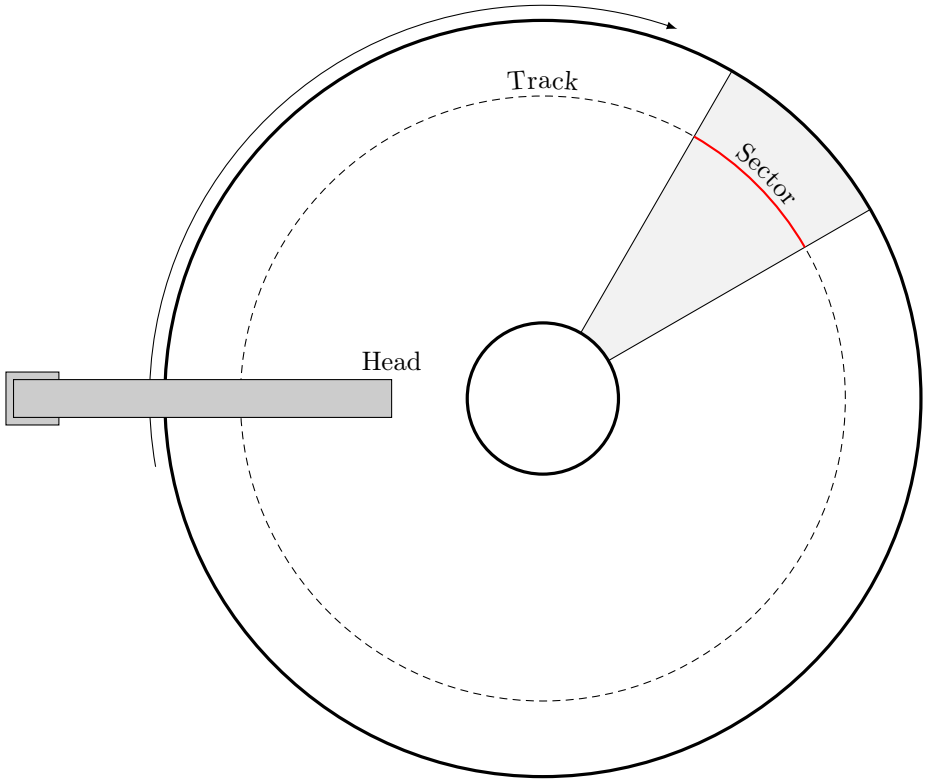
- DVDs
- CDs

Usually, the closer the storage is to the CPU, the faster, but smaller and more expensive it is:



Type	Capacity	latency
CPU	bytes	< 1 ns
Caches	kilo- / megabytes	< 10 ns
Main Memory	gigabytes	70 ns to 100 ns
Hard Disks	terabytes	3 ms to 10 ms
Tape Library	petabytes	*varies*

1.5 Magnetic Disks



A magnetic disk consists of a constantly rotating disk, and a *disk head*, which can perform read or write operations.

Disks can be subdivided into consentric *tracks*, and *sectors* on that tracks. Disks are managed in *blocks*. One block can cover one or more sectors. Writing only happens one block at a time.

Data can be read or written, iff the disk head is positioned over the corresponding block.

Usually big DBMS run on magnetic disk, because they are persistent. Note, that the disk block is not the same as a database page, even though they are sometimes both called blocks.

1.5.1 Data Access

There are several components that influnce the *access time*:

1. *seek time*: the disk head needs to be positioned on the right *track*
2. *rotational delay*: the disk head needs to wait for the correct *sector*
3. *transfer time*: finally the disk head can perform read or write operations

Sometimes one track is not enough, and another track also needs to be read or written to. In these cases the time to move between tracks is also an important factor.

1.5.2 Access Time

The access time t_a is defined as follows:

$$t_a = t_s + t_r + t_{tr}$$

t_s ... seek time
t_r ... rotational delay
t_{tr} ... transfer time

The time to move the head between two tracks t_{t2t} is called *track-to-track* time.

Sequential I/O is much faster than random I/O.

1.6 Solid State Drive

Have emerged as an alternative to hard disks. Solid State Drives (SSD) do not have any moving parts.

1.6.1 Technology

- usually interconnected flash memory cards
- no moving mechanical components, hence they are more robust
- more expensive

1.6.2 Performance

- much lower latency for random access
- random writes are typically slower

1.7 File Organization

1.7.1 File

A file can represent a table or an index. It consists of a sequence of records.

1.7.2 Records

A record is a single data item in a table or index. A record consists of a collection of related data values.

Each record has a *type* which describes what the allowed values for the different components of the record are. The record type also defines the type for the columns if it is part of a table. Records can have a *fixed length* or *variable length*. A variable length record can have variable length fields (*varchar* type) or optional fields (*null* values).

The standard types include:

- numeric types
- char
- boolean
- date
- time
- varchar
- null

For large data blocks, that are usually larger than a page, a different type is used:

- BLOB (binary large object)
- CLOB (character large object)

These items are stored in a separate pool of disk blocks, the record only contains the pointer to these blocks.

1.7.3 Record Blocking

Records are stored on a page.

The *Blocking Factor* bfr discribes the amount of records that fit on a page.

Assume a page of size B and a fixed length record of size R. The blocking factor bfr is equal to: $\text{bfr} = \lfloor \frac{B}{R} \rfloor$. Thus unused space on each page is $B - (\text{bfr} \cdot R)$ bytes.

1.7.4 Spanned Page Organization

Avoids unused page space, by allowing to store parts of a record on a page. To keep track where the other part of a record is stored, a pointer to the corresponding page is also stored.

This concept can be used to avoid unused page space or if the record size is larger than the page size: $R > B$.

1.7.5 Unordered vs. Ordered Records

Unordered Records	**Ordered records**
heap file, *pile file*	*sorted file*, *sequential file*
records are placed in the file in arbitrary order (append at end, or insert in free space)	records are sorted by some ordering field
inserting a new record is very efficient	inserting a record is expensive
searching for a record requires linear search	searching for a record is very efficient because of binary search
	deleting records is expensive

There are techniques to speed up the insertion or deletion of ordered records:

- insertion: place new records in unordered overflow file
- deletion: use delete marker and periodic file reorganization

1.8 Storage Information in PostgreSQL

- Check the size of tables and indices:
  - `SELECT pg_relation_size('title_basics');`
  - `SELECT pg_indexes_size('title_basics');`
  - `SELECT pg_relation_size('title_basics_idx1');`
- How many blocks are taken up by the table?
  - `SELECT relpages FROM pg_class WHERE relname = 'title_basics';`
  - `SHOW block_size;`
- Query the average blocking factor:
  - `SELECT reltuples/relpages FROM pg_class WHERE relname = 'title_basics';`

2 Indexing

2.1 Primary Index

In the *primary index* the file and the index are ordered based on the *ordering key field*. While the file contains the full records the index only contains the key value along with a pointer. The pointer can either be:

- a physical address to a record, a logical address, or
- an *anchor*.

An anchor is a pointer to the first record of a block. The *anchor record* is the first record of a block.

2.1.1 Data Access

Index entries are much smaller than data records, thus more index entries can be saved per page than data records. Binary search is also faster on the index than the file.

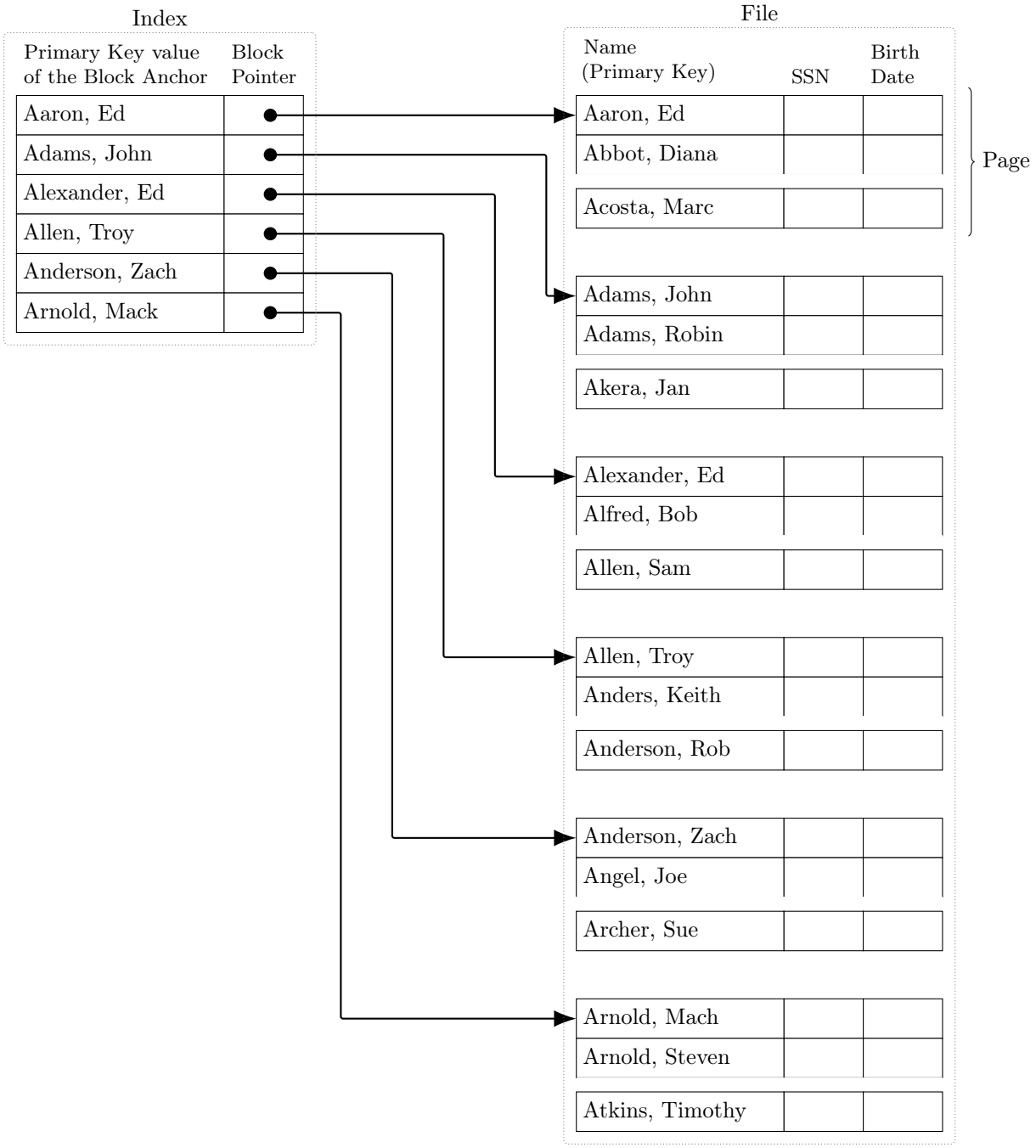


Figure 1: Primary Index Example

2.2 Clustering Index

In the *clustering index* the file and the index is also ordered on a specific field, just like in the primary index. But this time, we asume, that the field is not unique. Thus we order the index on an *ordering non-key field*.

The block pointer now point to the anchor record of the block, where the given value first shows up.

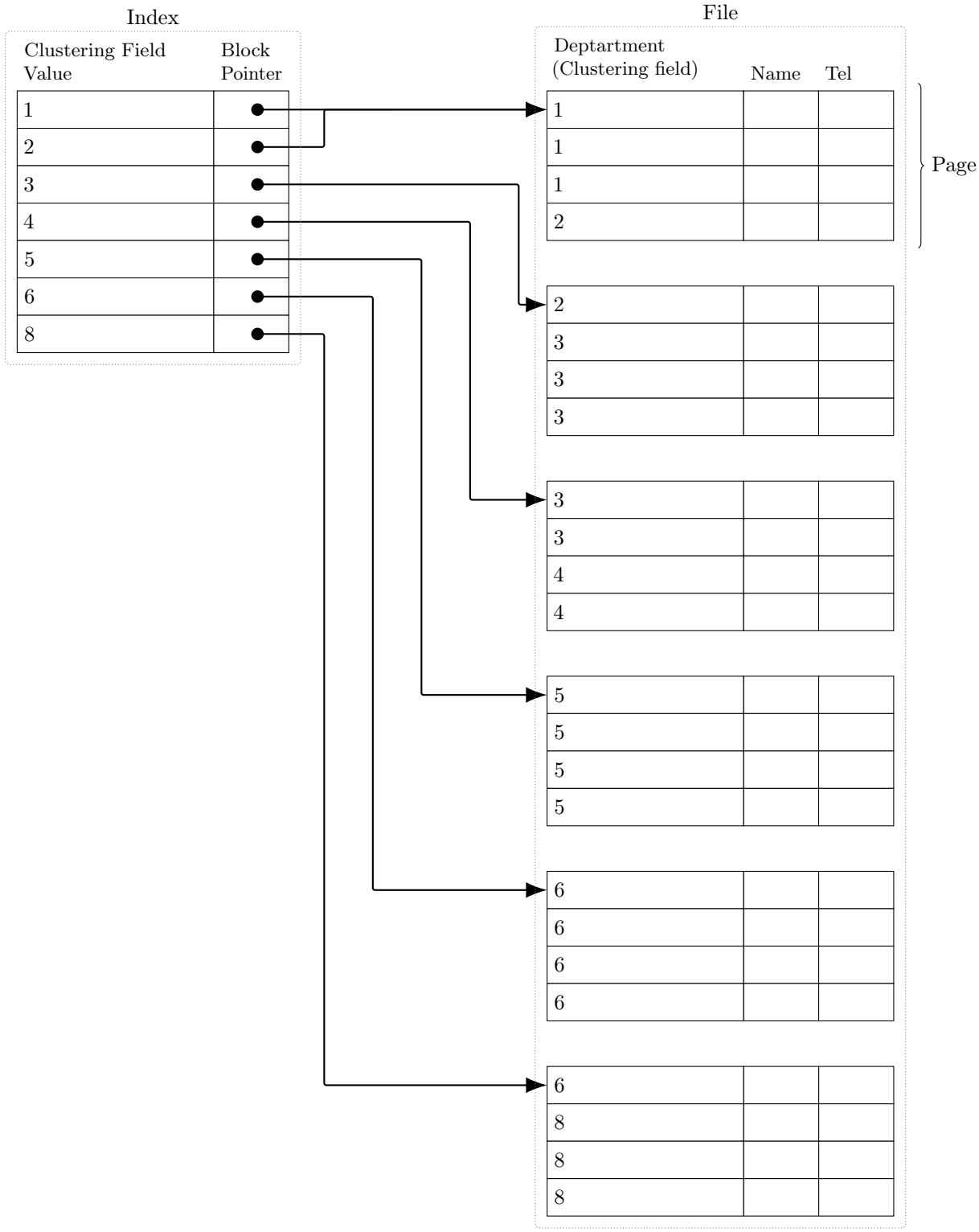


Figure 2: Clustering Index Example

2.3 Secondary Index

In the *secondary index* the index is ordered on an *ordering field*, but the file might not be ordered, or ordered by a different field.

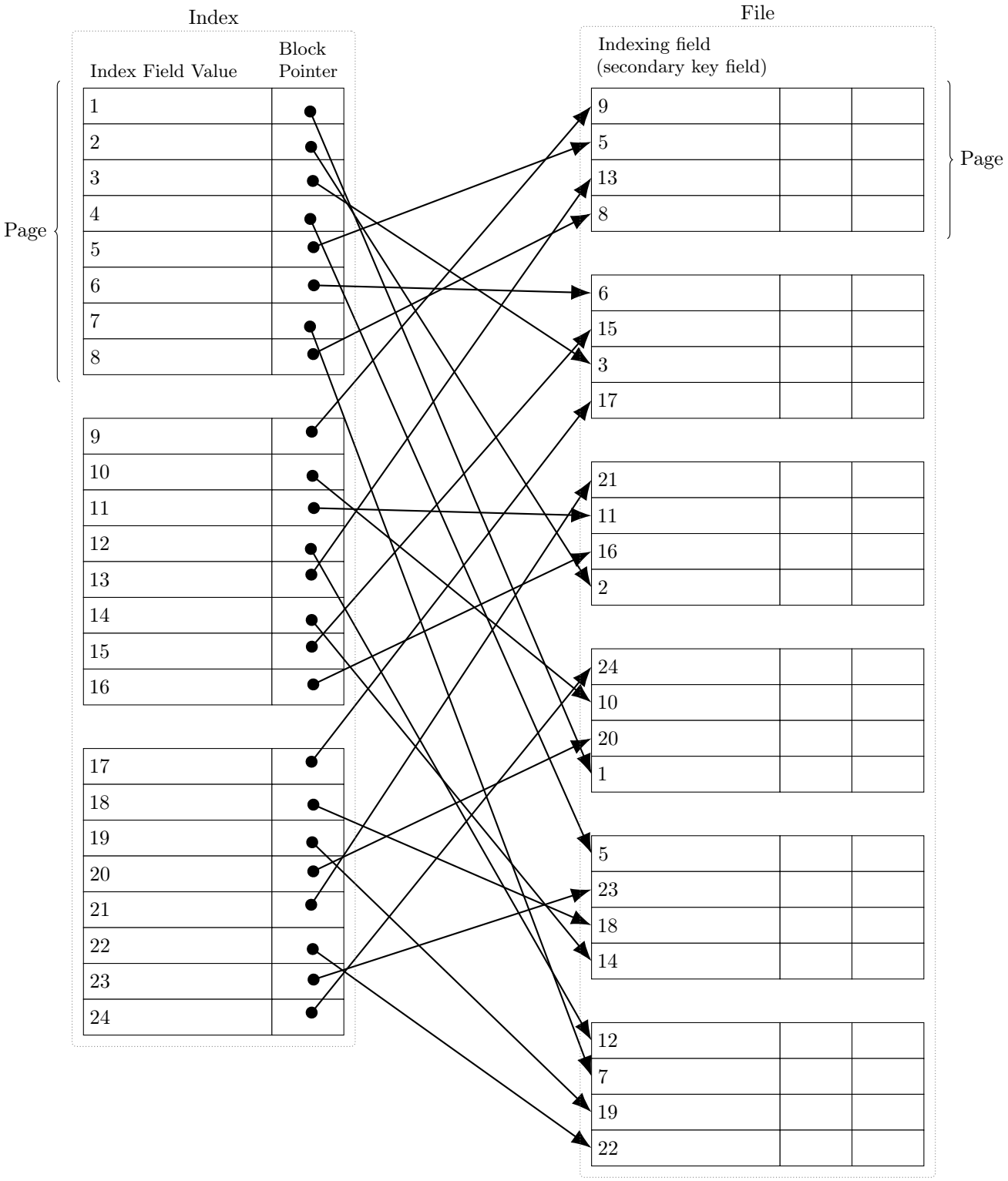


Figure 3: Secondary Index Example

2.4 Summary

Name	Ordering Type	Index Ordering	File Ordering
Primary Index	key field	ordered	ordered
Secondary Index	non-key field	ordered	not necessarily ordered
Cluster Index	ordering field	ordered	ordered

Table 1: Indices Summary

2.5 Overflow Pages

Overflow pages are appended at the end of existing pages. They can usually only be search with linear search. An increase in overflow pages often means a decrease in search performance.

If the number of overflow pages is too large, re-indexing is required.

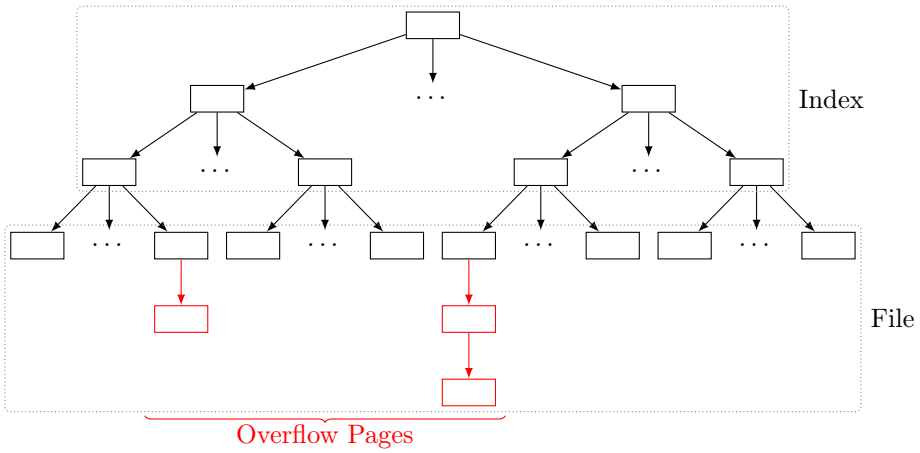


Figure 4: Overflow Pages

2.6 Multi-Level Index

Let b_f be the number of blocks (pages) for a file f. Then, the number of blocks b_i for the corresponding index i will be much smaller: $b_i \ll b_f$.

To binary search for an entry in a index, around $\log_2 b_i$ block accesses are needed.

If the file is huge, the index will also be big. It is possible, that the index alone is to large to be performant. To combat that, another index for the ordered index might be introduced. This can be done until the last index fits into a single page. This greatly reduces the accesses needed.

Example Suppose that we have 4 kB pages and that each index entry requires 10 B (4 B for an integer key field and 6 B for the pointer). Then, we have a blocking factor bfr of $\lfloor \frac{4096\,\mathrm{B}}{10\,\mathrm{B}} \rfloor = 409$ index entries per page. That means, each index entry is responsible for 409 other entries. Thus, the disk I/O is $t = \lceil \log_{409} b_i \rceil$.

2.6.1 ISAM (Index Sequential Access Method)

This indexing method was used in the MySQL implementation.

Single Level ISAM contains a primary index on an ordered file. Each index entry consists of (a) a key value, and (b) a pointer to the data file page.

Searching Single Level ISAM is done by using binary search on the index. ISAM is particularly usefull for range queries. For large indices, multilevel indexing is used.

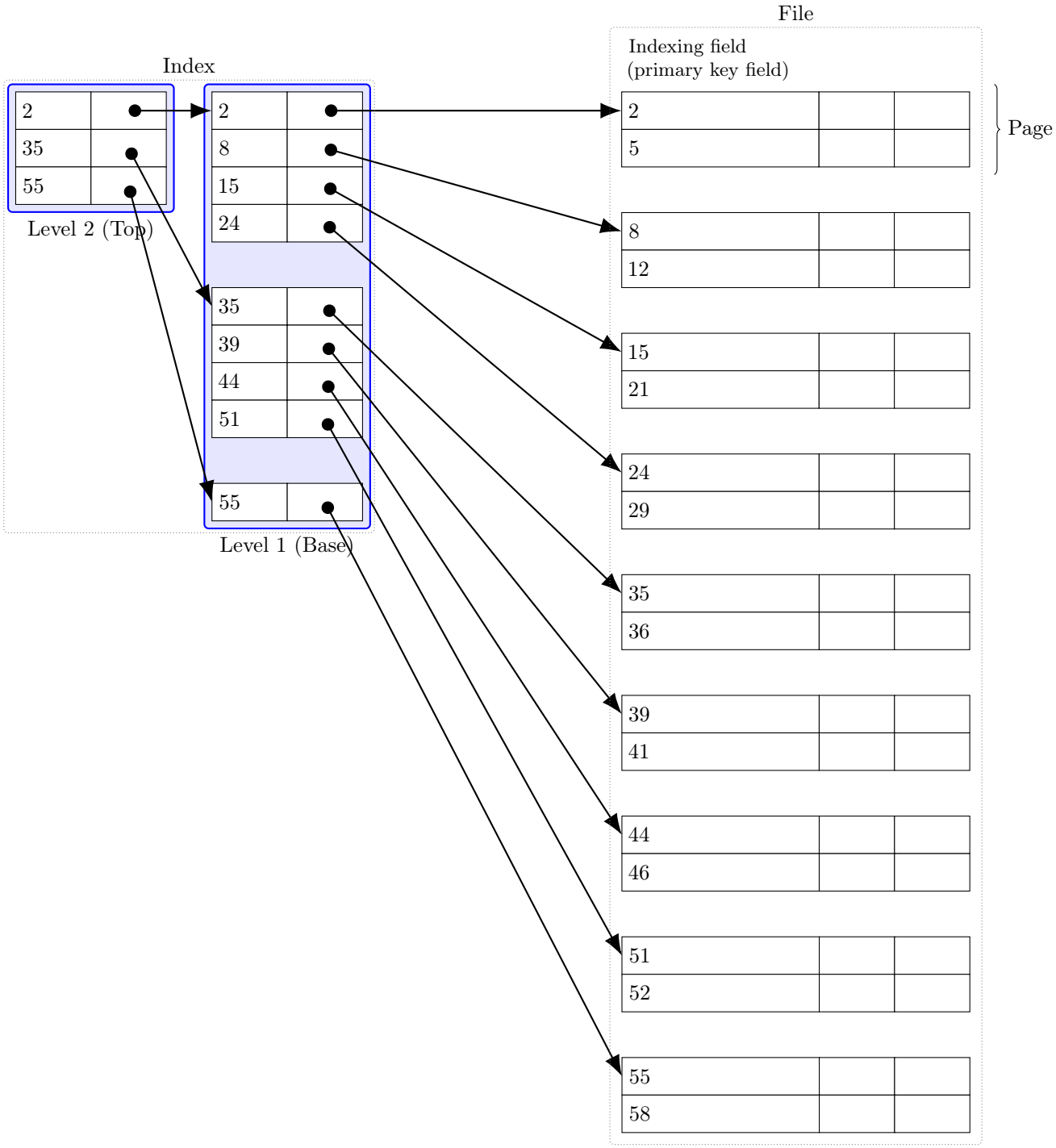


Figure 5: Multilevel Index Example

Strengths

- simple
- well suited for range queries

Weaknesses

- problems with maintaining the order
- problem with inserts
  - Data records are only inserted on the intended page if enough space is left, otherwise an overflow

page is appended.

- problem with deletes
  - Key value on higher level of the index may no longer apppear on the leaf level.
- over time, the search performance of ISAM can degrade, if too many overflow pages exist

In conclusion, ISAM is best suited for static data.

2.6.2 B-Tree and B$^+$-Tree Index

- *B$^+$-Tree Index*
  - leaf nodes are connected to form a doubly linked list
  - leaves may contain actual data records, e.g.:
    - primary index
    - clustering index
  - or pointers to data pages
- *B-Tree Index* inner nodes contain
  - data records, or
  - data pointers

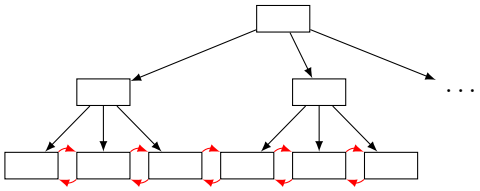


Figure 6: B$^+$-Tree Index

In practice mostly B$^+$-Trees are used. But in PostgreSQL, the B$^+$-Tree strategy is refered to as “btree”!

Characteristics

- a B$^+$-Tree index is always *balanced*
  - this means, that the distance from each leave node to the root node is the same
- never needs overflow pages
- efficient (log-time) in
  - search
  - insert
  - delete
- all nodes, except for the root node, have at least 50 % occupancy

Definition The definition of a B$^+$-Tree index of order p is as follows.

- each internal node is of the form:

$$\langle P_1, K_1, P_2, K_2, P_3, K_3, \ldots, K_{q-1}, P_q \rangle$$

  where $q \leq p$, each P_i is a tree pointer and K_i is a key value.
- within each node it holds that:

$$K_1 < K_2 < K_3 < \cdots < K_{q-1}$$

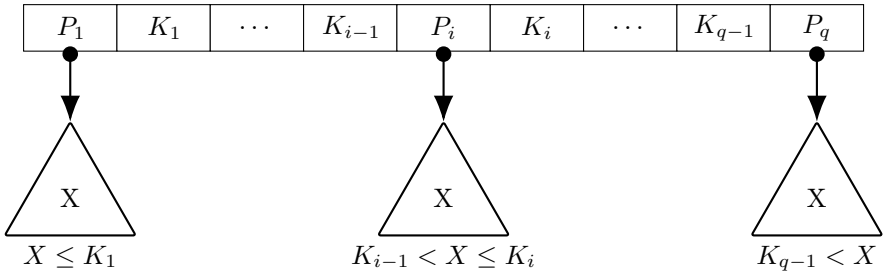
- each pointer P_i points to a subtree with key values
  - $< K_1\,, \quad$ for $1 = 1$
  - $\in [K_i, K_{i+1})\,, \quad$ for $1 < i < q$

  - $\leq K_{q-1}$, for $i = q$
- each node has at least $\frac{p}{2}$ entries
- leaf nodes contain pointers to the records or the actual records themselves
  - possible layouts of leaf nodes:
    * *pointer*

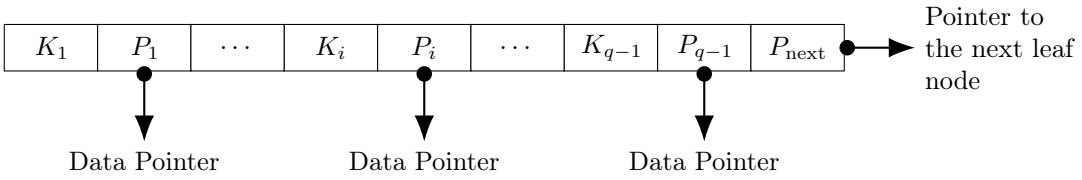
$$\langle\langle K_1, \text{RID}_1\rangle, \langle K_2, \text{RID}_2\rangle, \ldots, \langle K_q, \text{RID}_q\rangle\rangle$$

    * *data*

$$\langle\langle K_1, \text{record}_1\rangle, \langle K_2, \text{record}_2\rangle, \ldots, \langle K_q, \text{record}_q\rangle\rangle$$



(a) Layout of an inner node



(b) Layout of a leaf node

Figure 7: Layout of a B^+-Tree index

Insert Operation of a record with the key value k:

Insert Operation

```
search for the leaf page b where value k should be inserted
if the page has space #entries < p then do
    insert k
else if the page is full #entries = p then do
    split page b into two pages b and b'
    determine a key value k' that separates b and b' in two pages with p/2
    ↪ entries each
    insert key value k' at parent with the new pointer to b'
    if parent has no space then do
        recursively repeat splitting
    end
end
```

Splitting the page can propagate up to the root, leaving it with $q < \frac{p}{2}$, which is legal. The number of splitting operations is *logarithmically bound*.

Delete Operation of a record with the key value k:

Delete Operation

```
search for the leaf page b where value k is stored
delete value k
if the result is an underful page (#entries < p/2) then do
    try balancing with a neighbouring node (shift one key value from a
    ↪ neighbour to page b and adjust parent)
    if this would lead to an underful neighbour then do
        merge the two nodes
        delete key entry and pointer in the parent
    end
    recursively repeat deletion at the parent page
end
```

In practice no deletion is carried out, rather the record is *marked as deleted*. Thus periodic index reorganization is required. Additionally, only leaf nodes are merged.

2.7 Hash-Based Index

Static Hashing for an attribute A works as follows:

- allocate N disk pages (*buckets*)
- use a hash function to map each possible value of A to an integer value in $[0, N-1]$
- each bucket has a pointer to a chain of overflow pages

Characteristics

- hash index in principle unbeatable for equality-based queries
- performance degrades as the number of overflow pages grows
- useless for range queries

Solution to the problem of the overflow pages:

- *static hashing*
  - requires periodic index reorganization
- *extendible hashing*
  - use a directory of pointers to the buckets
  - increase the directory and split only the buckets that overflow

2.8 Bitmap Index

A bitmap is a vector of bits with a dimension of number of rows in a table. Each bitmap is created for only one attribute.

Bitmaps are particularly well-suited for

- set operations
- multi-key search
- counting rows

For the table 2 the bitmap index for the attribute *Sex* would be:

M	F
10100110	01011001

and for the attribute *Zipcode* it would be:

19046	30022	94040
00101100	01010010	10000001

Row ID	Employer ID	Name	Sex	Zipcode
0	51024	Bass	M	94040
1	23402	Clarke	F	30022
2	62104	England	M	19046
3	34723	Ferragamo	F	30022
4	81165	Gucci	F	19046
5	13646	Hanson	M	30022
6	12675	Marcus	M	30022
7	41301	Zara	F	94040

Table 2: Example Table

2.9 Multi-Key Index

- an index can be created on a combination of fields
- results in lexicographical ordering of the entries
- used for frequent queries with equality conditions on several fields
- combine attributes that are often searched for together to possibly allow for *index-only search*

2.10 Function-Based Index

- first apply a function f to a field or a combination of fields and then build an index on the result
- for frequent queries on values $f(V)$ for some function f applied to the value V of some field. For example in a case like this `SELECT ... WHERE f(attr) = x` the optimizer cannot use an index for the attribute `attr`. Another example would be a case insensitive index:
  `CREATE INDEX upper_idx ON Employee (UPPER(name))`

2.11 Summary

2.12 Index Types in PostgreSQL

- standard index types
  - B^+-Tree index (`BTREE`)
  - Hash-based index (`HASH`)
  - Bitmap cannot be requested by the user
- further index types
  - for specific purposes such as text search, geometric applications, etc.
  - GIN (Generalized Inverted Index)
  - GiST (Generalized Search Tree)
  - SP-GiST (Space-Partitioned GiST)
  - BRIN (Block Range Indexes)

2.13 Index Creation in PostgreSQL

- automatically created indices
  - for each primary key
  - for attributes with the `UNIQUE` constraint
- commands
  - `CREATE INDEX` (default type is a B^+-Tree)
  - `CLUSTER`
  - `REINDEX` reorganisation of index

To create a B^+-Tree index the following can be used:

```
CREATE INDEX byear_btree ON name_basics [USING BTREE] ("birthYear");
```

To create a hash index the following can be used:

```
CREATE INDEX byear_hash ON name_basics USING HASH ("birthYear");
```

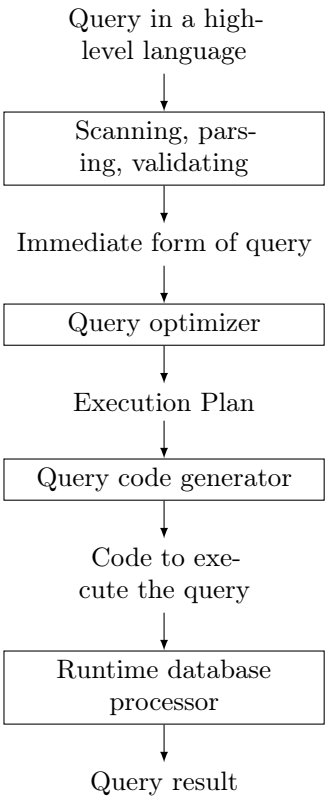
To get the size of an index the following can be used:

```
SELECT pg_relation_size($idxname);
```

3 Optimization

3.1 Query Evaluation Process

1. the *scanner* identifies the query tokens
2. the *parser* checks the query syntax
3. the *validation* checks all relations and attribute names
4. query tree creation
5. *query optimization*
   - for SQL
     - based on relational algebra expression
   - logical optimization
     - transformation of relational algebra (RA) expression independently of database statistics (heuristic rules)
   - physical optimization
     - based on cost model
     - determine "cheapest" plan
6. *execution strategy* or *query execution plan* (QEP) devised



3.2 Translation of SQL Query

Usually, a query will not be optimized as a whole, but rather split into different *query blocks*. A query block:

- is a basic unit that can be translated into algebraic expressions
- contains a single `SELECT`-`FROM`-`WHERE` expression
- may contain `GROUP BY` and `HAVING`

A query block is then translated into a relational algebra expression.

The relational algebra is usually extended (Extended RA) by other operations:

- sorting
- duplicate elimination
- semi-join

- anti-join
- aggregate functions
  - max
  - min
  - sum
  - count
  - average

Example for query blocks:

Example

```
SELECT name
FROM employee
WHERE salary > (
    SELECT MAX(salary)
    FROM employee
    WHERE department_number = 5
);
```

The *inner block* would be

Inner Block

```
SELECT MAX(salary)
FROM employee
WHERE department_number = 5
```

and consequently the *outer block* would be

Outer Block

```
SELECT name
FROM employee
WHERE salary > c
```

3.3 Relational Algebra Expressions

3.3.1 Basic Expressions

- relations of databases
- constant relations

3.3.2 Operations

3.3.3 Set Semantics vs. Bag Semantics

- *Set Semantics*
  - assumed in relational algebra
  - relations are *sets* of tupels (no duplicates)
- *Bag Semantics*
  - used in SQL
  - relations are *multisets* of tupels (duplicates allowed)

Table 3: Relational Algebra Operations

Name	Math
Selection	$\sigma(R)$
Projection	$\pi_S(R)$
Cartesian Product	$R_1 \times R_2$
Renaming	$\rho_V(R)$, $\rho_{A \leftarrow B}(R)$
Union	$R_1 \cup R_2$
Set Difference	$R_1 - R_2$

(a) Basic Relational Algebra Operations

Name	Math
Intersection	$R_1 \cap R_2$
Division	$R_1 \div R_2$
Various Joins	$R_1 \bowtie R_2$
Semi Join Left	$R_1 \ltimes R_2$
Semi Join Right	$R_1 \rtimes R_2$
Outer Join Left	$R_1 \mathbin{⟕} R_2$
Outer Join Right	$R_1 \mathbin{⟖} R_2$
Outer Join Full	$R_1 \mathbin{⟗} R_2$

(b) Compound Relational Algebra Operations

3.3.4 Selection

The selection operation selects a subset of tupels (rows) from a relation based on a *selection predicate* (filter condition).

The selection predicate is build from comparison operators and combined by logical connectives.

Example

$$\sigma_{\text{department\_number}=4}(\text{employee})$$
$$\sigma_{\text{salary}>30000}(\text{employee})$$

3.3.5 Projection

The projection operation selects a subset of attributes (columns) from a relation. This operation might lead to duplicates, which have to be eliminated in case of set semantics.

Example

$$\pi_{\text{name,salary}}(\text{employee})$$
$$\pi_{\text{sex,salary}}(\text{employee})$$

3.3.6 Common Properties of Set Operations

- the two operants must be *type compatible*
  - the have the same number of attributes and each pair of corresponding attributes has compatible domains
- the `UNION` may produce duplicates
  - they have to be eliminated, even in SQL
  - in SQL with `UNION ALL`, duplicates survive
- `INTERSECT`, `EXCEPT` (sometimes called `MINUS`) do duplicate elimination

3.3.7 Expressing Intersection by other Operations

$$R \cap S = R - (R - S)$$

3.3.8 Cartesian Product (Cross Product)

The *cartesian product* combines every tupel of the first relation with every tupel of the second relation. The attributes of the resulting relation consists of the (disjoint) union of the attributes of the two relations.

Usually, the cartesion product is meaningless without a following selection. Any cartesion product with a selection, whose predicate refers to attributes from both operants, is better expressed as a *join*.

3.3.9 Division

Let R be a relation with the attribute set $X \cup Y$ and S be a relation with attribute set Y, then $T(X) = R(X,Y) \div S(Y)$ is a relation with attribute set X, and $T(X)$ contains all tuples t such that $(t,s) \in R$.

The *division* operation is defined as:

$$R(X,Y) \div S(Y) = T(X)$$

Intuitively, the result of $R(X,Y) \div S(Y)$ contains those X-values which, when combined via the cartesian product with tupels from S, only yield tupels in R.

Division can also be expressed by basic operations:

$$\begin{aligned} R(X,Y) \div S(Y) &= \pi_X(R) - \pi_X\left((\pi_X(R) \times S) - R\right) \\ &= \pi_X(R \cap (\pi_X(R) \times S)) \end{aligned}$$

3.3.10 Natural Join

Let $R(X,Y)$ and $S(Y,Z)$ be two relations, such that $X \in R$, $Y \in R \wedge Y \in S$ and $Z \in S$. Then,

$$T = R \bowtie S$$

is a relation with attribute set $X \cup Y \cup Z$ and $t \in T$ iff $\pi_{X \cup Y}(t) \in R$ and $\pi_{Y \cup Z}(t) \in S$.

The natural join can also be expressed by basic operations:

$$R \bowtie S = \pi_{X \cup Y \cup Z}(\sigma_{R.Y = S.Y}(R \times S))$$

3.3.11 Theta Join (Non-Equi-Join)

Let $R(X,Y)$ and $S(X,Y)$ be two relations, then,

$$T = R \bowtie_\theta S$$

is a relation with attribute set $U \cup V \cup X \cup Y$ and $t = (r,s) \in T$ iff $r \in R$, $s \in S$ and (r,s) satisfies condition θ

The theta join can also be expressed by basic operations:

$$R \bowtie_\theta S = \sigma_\theta(R \times S)$$

3.3.12 Left Semi Join

The left semi join chooses those tuples from R, which have a join-partner in S. The right semi join works analogously.

$$R \ltimes S = \pi_R(R \bowtie S)$$

Typical SQL expressions, that result in semi joins are:

- `EXISTS`
- `IN`
- `ANY`

Semi Join Example

```
SELECT COUNT( * )
FROM employee E
WHERE e.department_number IN (
    SELECT d.department_number
    FROM department d
```

```
    WHERE d.city = Vienna
);
```

In this example R would be the outer block, while S would be the inner block.

3.3.13 Anti Join

The anti join is effectively the opposite of the semi join. Again, the right anti join works analogously to the left anti join. The left anti join selects all those tuples from R which do *not* have a join partner in S:

$$R \overline{\ltimes} S = R - (R \ltimes S)$$

Typical SQL expression, that result in anti joins are:

- `NOT EXISTS`
- `NOT IN`
- `ALL`

Anti Join Example

```
SELECT COUNT( * )
FROM employee e
WHERE e.department_number NOT IN (
    SELECT d.department_number
    FROM department d
    WHERE d.city = Vienna
);
```

3.3.14 Left Outer Join

The left outer join tries to join R with S, but also adds those tuples from R which do not have a join partner in S. The missing fields are filled with `NULL`, such that the number of `NULL` elements in N corresponds to the number of those attributes in S that are not involved in the natural join $R \bowtie S$.

$$R \overline{\bowtie} S = R \bowtie S \cup (R \overline{\ltimes} S) \times \{(\overbrace{\texttt{NULL}, \ldots, \texttt{NULL}}^{N})\}$$

3.3.15 Full Outer Join

The full outer join tries to join R and S, but also adds those tupels from R and S, which have no join partner.

3.4 Relation Algebra Equivalences

3.4.1 Cascading Selections

$$\sigma_{c_1 \wedge c_2 \wedge \cdots \wedge c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))\ldots))$$

3.4.2 Commuting Selections

$$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$$

3.4.3 Cascading Projections

Provided that $L_1 \subseteq L_2 \subseteq \cdots \subseteq L_n$.

$$\pi_{L_1}(\pi_{L_2}(\ldots(\pi_{L_n}(R))\ldots)) \equiv \pi_{L_1}(R)$$

This rule is often applied from right to left, meaning, that instead of one large projection, it is split up into multiple subsets of projections.

3.4.4 Commuting Selection and Projection

Provided that all attributes used in the selection occur in L

$$\sigma_c(\pi_L R) \equiv \pi_L(\sigma_c(R))$$

3.4.5 Commuting Cartesian Product

$$R \times S \equiv S \times R$$

3.4.6 Commuting Set Operations

$$R \cup S \equiv S \cup R$$
$$R \cap S \equiv S \cap R$$

3.4.7 Commuting Join Operations

$$R \bowtie S \equiv S \bowtie R$$
$$R \bowtie_\theta S \equiv S \bowtie_\theta R$$

3.4.8 Commuting Selection and Join

Provided that the selection only uses predicates in one of the relations, i.e. R:

$$\sigma_c(R \bowtie_\theta S) \equiv \sigma_c(R) \bowtie_\theta S$$

3.4.9 Commuting Projection and Join or Cartesian Product

Assuming, that relations R and S have no attributes in common.

- if $L = L_1 \cup L_2$, such that all attributes in L_1 are from R and all attributes in L_2 are from S, and the join predicate C only uses the attributes in L:

$$\pi_L(R \bowtie_C S) \equiv \pi_{L_1}(R) \bowtie_C \pi_{L_2}(S)$$

- if $L = L_1 \cup L_2$, such that all attributes in L_1 are from R and all attributes in L_2 are from S, and the join predicate C uses the attributes in L plus further attributes M_1 from R and M_2 from S:

$$\pi_L(R \bowtie_C S) \equiv \pi_L(\pi_{L_1,M_1}(R) \bowtie_C \pi_{L_2,M_2}(S))$$

3.4.10 Associativity of Cartesian Product

$$(R \times S) \times T \equiv R \times (S \times T)$$

3.4.11 Associativity of Set Operations

$$(R \cup S) \cup T \equiv R \cup (S \cup T)$$
$$(R \cap S) \cap T \equiv R \cap (S \cap T)$$

3.4.12 Associativity of Join Operations

$$(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$$
$$(R \bowtie_\theta S) \bowtie_\theta T \equiv R \bowtie_\theta (S \bowtie_\theta T)$$

3.4.13 Replace Cartesian Product with Selection by Join

Provided that the selection predicate C is a condition of the form $A\,\theta\,B$, where A is an attribute in R and B is in S.

$$\sigma_C(R \times S) \equiv R \bowtie_C S$$

3.4.14 Distributivity of Selection with Set Operations

$$\sigma_C(R \cup S) \equiv \sigma_C(R) \cup \sigma_C(S)$$
$$\sigma_C(R \cap S) \equiv \sigma_C(R) \cap \sigma_C(S) \equiv \sigma_C(R) \cap S$$
$$\sigma_C(R - S) \equiv \sigma_C(R) - \sigma_C(S) \equiv \sigma_C(R) - S$$

3.4.15 Distributivity of Projection and Union

$$\pi_L(R \cup S) \equiv \pi_L(R) \cup \pi_L(S)$$

3.4.16 De Morgan's Laws for Join and Selection Predicates

$$\neg(C_1 \wedge C_2) \equiv (\neg C_1) \vee (\neg C_2)$$
$$\neg(C_1 \vee C_2) \equiv (\neg C_1) \wedge (\neg C_2)$$

3.5 Rule-Based Optimization

The idea of rule-based optimization, which is logical optimization, is (i) to apply equivalence-preserving transformations, and (ii) to use heuristic rules which yield a "simpler" relational algebra expression.

Rule-based optimization says nothing about the concrete implementation of each operator from the extended relational algebra. The goal is to make intermediate results as small as possible.

This can be done by:

- replacing cartesian producs with selections by joins
- applying selections or projections as soon as possible
- combine selections or projections to avoid double I/O access

A `SELECT`-`FROM`-`WHERE` query might get translated canonically like this:

1. *cartesian product* of alle relations in the `FROM` clause
2. *selection* with predicate in the `WHERE` clause
3. *projection* to attributes in the `SELECT` clause

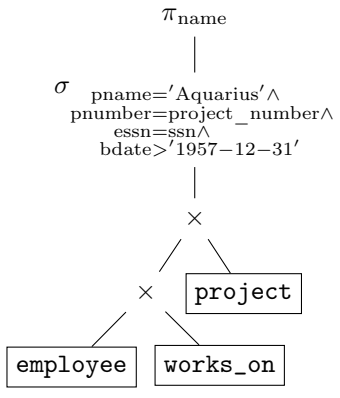
Typical transformations are:

- cascading selections
- pushing selections (applying selections as early as possible)
- replacing a cartesian product with selection by a join
- carry out most selective selections or joins first
- project out attributes that are not needed further up in the tree
- identiy groups of operations that can be executed together by a single algorithm

3.5.1 Example

Example

```
1 SELECT e.name
2 FROM employee e, works_on w,
    ↪ project p
3 WHERE p.pname = 'Aquarius' AND
4         p.pnumber = w.
    ↪ project_number AND
5         e.essn = w.ssn AND
6         e.bdate > '1957-12-31';
7
```



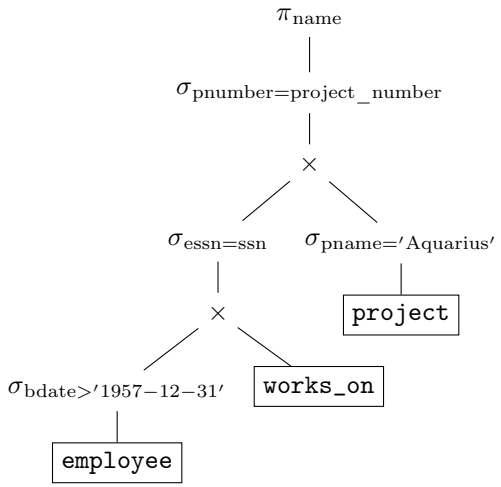


Figure 8: Step 1, split the big selection statement, and move it down the tree.

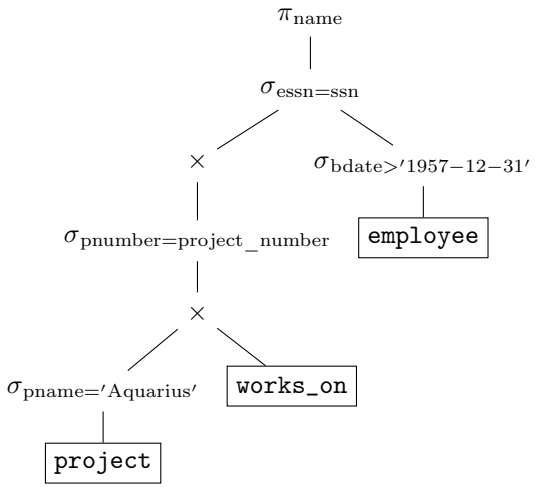


Figure 9: Step 2, apply the more restrictive selections first.

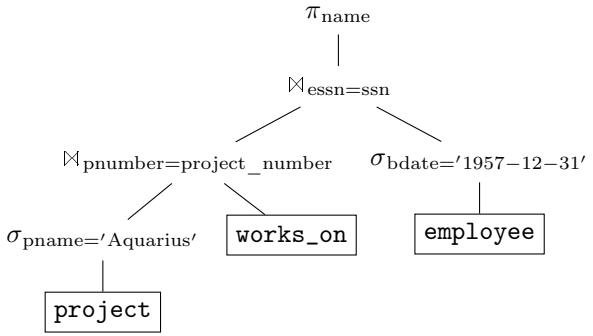


Figure 10: Step 3, replace cartesian producs with selections by joins.

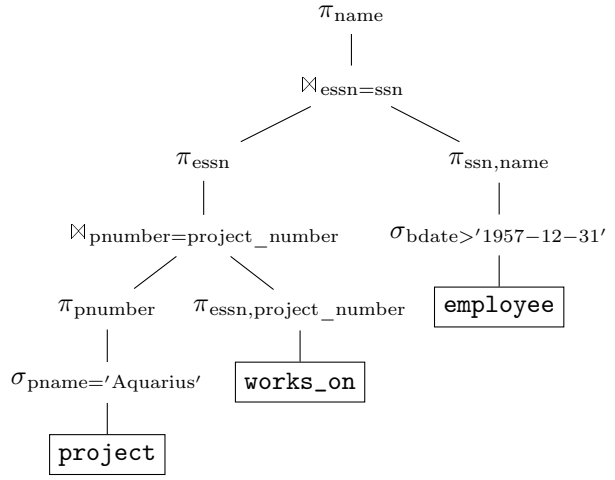


Figure 11: Step 4, move projections down the query tree.

3.6 Cost-Based Optimization

The DBMS assigns a *cost* – some metric that measures the required resources – to each query plan, and the goal is to find the query plan with the minimal estimated cost. After the query tree of the relational algebra is fixed, there are still a lot of open questions. A query plan (query execution strategy) answers these questions:

- choose one (out of many) equivalent query trees
- choose for each leaf node an access method to this relation
- choose for each inner node (relational operator) an implementation

The choices made by the query plan define the search space.

3.6.1 Cost of a Query Plan

- main cost components
  - disk I/O: cost of reading / writing data block from / to the disk
  - CPU cost: cost of performing in-memory operations
  - memory usage: number of buffer frames needed
  - communication costs: important for distibuted databases
- cost function
  - assigns a numeric value depending on the estimated consumption of the above resources to every query plan
  - optimization goal is the minimization of this function
  - typical cost functions only consider a *single cost factor*
    - for big databases: disk I/O
    - for small databases: CPU cost

3.6.2 Impact on the Cost Function

- Aspects, that affect the cost function
  - size of the input relations
  - "sortedness" of input relations
  - presence / absence of indices
  - available space in the buffer

3.6.3 Cost Function

r_X ... number of records (tupels) in a relation X
b_X ... number of blocks occupied by relation X
bfr_X ... blocking factor in relation X
sel_X ... selectivity of predicate P (percentage of tupels satisfying this condition)
s_P ... selection cardinality of predicate P ($\text{sel}_P \cdot r_X$)

4 Evaluation of Relational Operators

4.1 Sorting

Sorting is an operation of the extended relational algebra, and can be requested by the user via a `ORDER BY` clause.

- small files
  - file fits entirely into memory
  - simply load the file and apply an efficient sorting algorithm (typically quicksort)
- large files

- file does not fit entirely into memory
- page-wise file organization has to be taken into account
- solution is *external sorting*

4.1.1 External Sorting

The idea of external sorting is to produce sorted subfiles (runs). These runs are then merged into bigger runs. The sorting is done, if only one run remains.

External sorting requires buffer space – ususally three buffer frames are sufficient – and profits from more buffer space.

External sorting can be split into two phases:

- *initialization*, and
- *merge phase*.

Initialization For this phase one buffer frame suffices.

Initialization

```
foreach page do
    read page into buffer
    sort page
    write page to disk
end
```

The pages, which are sorted in this phase are called *level 0 runs*.

Merge Phase For this phase three buffer frames suffice.

Merge Phase

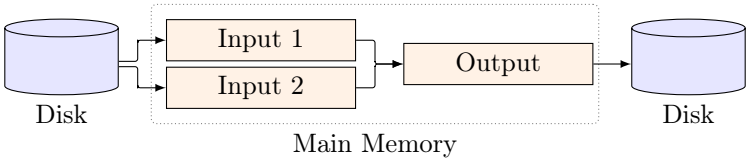
```
repeat until only one run is left do
    read 2 level i runs of length L into buffer
    merge them to a level i+1 run of length 2L
end
```

In principle, only three buffer frames are needed, one for each input page, and one for the resulting page.

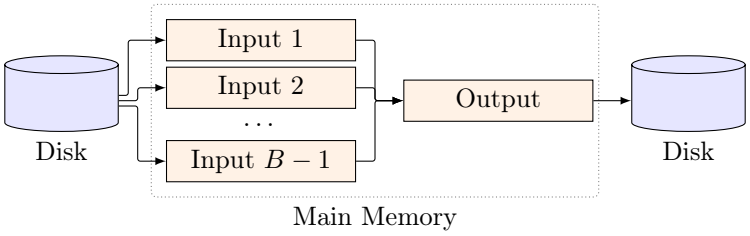
4.1.2 External Sorting with Realistic Buffer Sizes

In practice, more buffer frames than the minimum amount are used. For example, if B buffer frames are used, then the level 0 runs are no longer of length 1, but of length B, because in the initialization phase, B pages can be read and sorted at once.

For the merge phase, we can merge $B-1$ runs at once.



(a) External Sorting with 3 Buffer Frames



(b) External Sorting with B Buffer Frames

Figure 12: Memory Scheme of External Sorting

4.1.3 Disk I/O

Two-Way Merge Sort For a two-way merge sort with 3 buffer frames, and b number of pages. Each pass reads and writes each page exactly once.

The *number of passes* is the number of passes for each phase of the sorting algorithm, and thus:

$$\underbrace{1}_{\text{initialization}} + \underbrace{\lceil \log_2(b) \rceil}_{\text{merge phase}}$$

The *total cost* is calculated by the read and write, which happens per pass, times the number of passes:

$$\underbrace{2 \cdot b}_{\text{read \& write}} \cdot \underbrace{(1 + \lceil \log_2(b) \rceil)}_{\text{number of passes}}$$

Multi-Way Merge Sort The multi-way merge sort is analogous to the two-way merge sort, but with B buffer frames.

The *number of level* 0 *runs* is:

$$\left\lceil \frac{b}{B} \right\rceil$$

The *number of passes* is:

$$1 + \left\lceil \log_{B-1} \left(\left\lceil \frac{b}{B} \right\rceil \right) \right\rceil$$

The *total cost* is:

$$2 \cdot b \cdot \left(1 + \left\lceil \log_{B-1} \left(\left\lceil \frac{b}{B} \right\rceil \right) \right\rceil \right)$$

4.2 Selection

Basic search methods can be utilized for a selection with a condition of the type att = val. We assume:

b ... number of pages
sel ... selectivity of the selection predicate
r ... number of records
bfr ... blocking factor

The costs of the following searches exclude the output cost of the results. The output costs of the results is always the number of pages to be written:

$$\left\lceil \frac{\text{sel} \cdot r}{\text{bfr}} \right\rceil$$

4.2.1 Linear Search

Linear search (sequential scan) is a brute force algorithm, because is scans the whole file.

If the attribute *att* is not unique or if the value *val* does not exist, the total cost of the disk I/O is the number of pages:

$$b$$

On average, if the attribute *att* is unique, the average cost is:

$$\frac{b}{2}$$

4.2.2 Binary Search

Binary search can be utilized on an ordered file. The first matching tuple can be found after

$$\lceil \log_2(b) \rceil$$

reads. Since the file is ordered, we can then sequentially scan all the subsequent pages until the predicate does not hold anymore. Thus, the cost is:

$$\log_2(b) + \left(\left\lceil \frac{\text{sel} \cdot r}{\text{bfr}} \right\rceil - 1 \right)$$

In this equation, $\text{sel} \cdot r$ is the amount of tupels that satisfy the search condition. This is divided by the blocking factor bfr, since reads are happening on a block basis.

For unique attributes this simplifies to:

$$\log_2(b)$$

4.2.3 Index Search

Can be utilized, if an index is available. Sometimes an *index-only-search* is possible, if the index contains all requested attributes.

The I/O costs depends on the type of index, and how many tuples should be retreived.

Retreiving One Tuple

- *B^+-Tree Index*:
  For a B^+-tree index with depth d the cost is

  $$d$$

  if the leaf nodes carry all requested attributes.

  If not, the cost also includes the access of the referenced record:

  $$d + 1$$

- *Hash Index*:
  For a hash index the cost ideally is:

  $$1$$

  But due to the overflow pages, a common assumption is, that the cost c is more than that:

  $$1 \leq c \leq 2$$

Retreiving Multiple Tuples

- *Clustered Index*:
  Using a clustered index, we first need to find the first tuple that matches, and then perform a sequential search, as long as the condition holds. For a cost c of finding the first matching tuple, the total cost is:

$$c + \left\lceil \frac{\text{sel} \cdot r}{\text{bfr}} \right\rceil$$

- *Unclustered Index*:
  Using an unclustered index, first find all the tuples $\langle \text{val}, \text{RID} \rangle$ which match the condition, then retreive all the relevant pages, based on the record ID (RID), and perform a sequential search on these pages.

  If the selectivity sel is realy low, a full sequencial scan might be preferable.

Ranged Queries Ranged queries of type $\text{att} \geq \text{val}_1 \wedge \text{att} \leq \text{val}_2$ is treated similar to a single equality query, with a non-unique attribute. In this case a hash index does not help.

4.2.4 Complex Selection Predicates

For complex selection predicates of the form

$$\text{att}_1 = \text{val}_2 \wedge \text{att}_2 = \text{val}_2 \wedge \ldots$$

we need to adjust the search There are different approaches:

- Using an individual index on the attribute att_i, retreive all records that satisfy the condition $\text{att}_i = \text{val}_i$, and check if other conditions are also satisfied.
- Use a composite index on several or all attributes att_i
- Carry out several searches, one for each $\text{att}_i = \text{val}_i$ clause. After that, comput the intersection of all the record IDs (RIDs). This can only be done, if the clauses are combined with the logical $\wedge$. If the search predicate contains logical $\vee$ conditions, fewer options are available. Either use a bitmap index for a union, or a full sequencial scan.

4.3 Joins

Here, we concentrate on *two-way equi-joins*:

$$R \bowtie_{A=B} S$$

for some attribute A in R and another attribute B in S.

4.3.1 Nested Loops Join

For each page p_R of the "outer" relation R iterate through all pages p_S of the "inner" relation S. While iterating, check if the join predicate holds.

Nested Loops Join

```
1 foreach page p_R of R do
2     foreach page p_S of S do
3         foreach tuple r ∈ p_R ∧ s ∈ p_S do
4             if s.B = r.A then
5                 res := res ∪ {(r, s)}
```

The cost is defined as:

$$b_R + (b_R \cdot b_S)$$

Again, this implementation only needs three buffer frames, one for R, one for S and one for the result.

4.3.2 Block Nested Loops Join

Using B number of buffer frames, read $B-2$ pages of R at a time and try to combine all records in these $B-2$ pages with the records of the current page of S.

The cost ist defined as:

$$b_R + \underbrace{\left\lceil \frac{b_R}{B-2} \right\rceil}_{\text{loop iterations}} \cdot b_S$$

In general, it is better to use the smaller of the two relations in the "outer" loop.

4.3.3 Index Nested Loops Join

This can be utilized if an index for the join attribute B of S exists. For every page p_R of the "outer" relation R and each tuple r of R on page p_R use the index of attribute B to retreive all tuples s in S with $s.B = r.A$.

The cost depends on the type of index and if the attribute B os a unique attribute in S. But for example, if a B^+-tree index with depth d for the unique attribute B is used, then the total cost consists of iterating over all tupels in R and performing an index lookup over each one:

$$b_R + (|R| \cdot d)$$

4.3.4 Sort-Merge Join

Suppose that the tuples of R are sorted on the attribute A and the tuples of S are sorted on B, we can scan the pages of A and B in this order and search for matching tuples.

If at least one of the attributes (A or B) is unique, then we need to scan each of R and S only once.

The cost if both R and S are sorted and at least one attribute is unique is:

$$b_R + b_S$$

If R and/or S are not sorted, the cost of sorting has to be added.

If neither A in R nor B in S is unique, then we have to compute the cross product $\sigma_{A=v}(R) \times \sigma_{B=v}(S)$ for each value v.

The DBMS can only rely on the "sortedness", if an index exist, or if the sorts the relations. Otherwise, it cannot say, if a relation is sorted.

4.3.5 Hash Join

Apply the same hash function to $R.A$ and $S.B$ then only pairs of tuples in the corresponding buckets can join. A hash join can be split up into two phases:

- partitioning (hashing)
- probing (joining)

Partitioning Suppose that we have B buffer frames. The hash function is needed for at most $B-1$ buckets.

Read one page p_R of R at a time and determine by the hash function for each tupel on page p_R the bucket in $[1, B-1]$. When a page for a bucket is full, empty it by writing to the disk. Then do the same with every page p_s of S.

The cost of partitioning is:

$$2 \cdot (b_R + b_S)$$

Probing For every bucket number $i \in [1, B-1]$ do the following: Let R_i and S_i denote the i-th bucket of R and S respectively. Suppose that the smaller of the two (e.g. R_i) fits into $B-2$ buffer frames. Then we load R_i into the buffer and read one page of S_i at a time and determine all matches between the tupels on the current page of S_i with all of R_i.

In total we need B buffer frames, one for S_i, one for the output, and $B-2$ for R_i.

The cost of probing is:

$$b_R + b_S$$

and the total cost of the hash join is:

$$2 \cdot (b_R + b_S) + (b_R + b_S) = 3 \cdot (b_R + b_S)$$

Because hash joining has a linear costs, it is often the first method that DBMSs choose.

If both R_i and S_i are too big for the buffer, either:

A: hash R_i and S_i recursively, or

B: resort to nested loops join of R_i and S_i, using the smaller of the two as outer relation.

4.3.6 Hybrid Hash Join

If the buffer is large enough, we can keep one or more buckets of the smaller relation (say R) in the buffer. For these buckets, we can skip writing to disk in the partition phase and the reading in probe phase for both R and S.

Suppose we can keep k out of m buckets of R in the buffer, and suppose that the m buckets if R and S are approximately of equal size ($\frac{b_R}{m}$ and $\frac{b_S}{m}$ number of pages each), the cost is:

$$\left(3 - 2 \cdot \frac{k}{m}\right) \cdot (b_R + b_S)$$

Hybrid hash join does require a lot of memory, specifically this unequality has to be satisfied:

$$B \geq k \cdot \frac{b_R}{m} + (m - k) + 1$$

In the special case where all of R fits into the buffer, where $k = m$ the cost reduces to:

$$b_R + b_S$$

4.4 Other Types of Joins

4.4.1 Theta-Join

Theta joins have more general join conditions. The comparison operators are different from equality:

$$R.A \circ S.B$$

where

$$\circ \in \{<, >, \leq, \geq, \neq\}$$

Theta joins can be implemented the following ways:

- (block) nested loops join is always applicable
- other implementations may also be applicable with $\circ = \leq$
  - index nested loops
  - sort merge join

Only hash joinging will not work, as it needs an equality operator ($\circ = =$).

4.4.2 Outer Join

The left- / right outer joins $R \ltimes\!\!\!\rightarrow S$ / $S \ltimes\!\!\!\leftarrow R$ can be computed by modifying one of the join methods:

- when using (Index / Block) nested loops choose R as the outer relation
- for all other join methods simply add the tupels of R without a join partner (padded with `NULL`s) to the result
- take the union of join and anti join

The full outer join $R \ltimes\!\!\!\rightarrow\!\!\!\leftarrow S$ can be implemented by applying

- sort-merge, or
- hash join

and account for the unmatched tuples in both relations R and S.

4.4.3 Semi Join

Can be computed by modifying one of the join methods. The search for join-partners stops as soon as one is found. In this case, add the tuple from the outer relation to the result.

4.4.4 Anti Join

Can be computed by modifying one of the join methods. The search for join-partners stops as soon as one is found. In this case, reject the tuple from the outer relation.

4.5 Other Operations

4.5.1 Union

Simply append files if no duplicate elimination is requested (through `UNION ALL`).

4.5.2 Intersection

The intersection without duplicate elimination (`INTERSECT`) is a special case of join. Specifically, an equi-join over all attributes.

4.5.3 Cartesian Product

Cannot do better than nested loops.

4.5.4 Set Difference

Without the duplicate elimination (`EXCEPT`) it is similar to a join. In principle, all join methods are applicable (like anti join), but sort merge join or hash join are preferable.

4.6 Duplicate Elimination

Duplicate elimination is not a relational algebra operation but may be required

- by the `DISTINCT` clause in the query, orange
- as default behaviour of set operations in SQL.

It is important to note, that duplicate eliminations are expensive and the `DISTINCT` clause should be avoided if not needed.

Duplicate elimination can be implemented via:

- Sorting, or
  - sort the file and eliminate consecutive duplicates
- Hashing
  - each record is hashed and inserted into a bucket in memory
  - before inserting a record into a bucket, check if an identical record already exists in this bucket

4.7 Aggregate Operations

The operations: `MIN`, `MAX`, `COUNT`, `SUM` and `AVG` can be computed by a table scan, or by an index scan if a suitable index exists and index-only scan is supported.

In case of a sorted file, or a B^+-tree index, the result for `MIN` and `MAX` can be obtained as the first / last record of the file, or by a single descent in the B^+-tree index, following the left / right most pointer.

In case of a dense index (one index entry for every record in the main file) the result for the `SUM` and `AVG` functions can be computed via an index scan. This is only possible, if index-only search is supported.

The function `COUNT` can also possibly be computed from the index only.

4.8 Group By

- aggregate functions must be applied separately to each group of tuples
- usual techniques are sorting or hashing according to the grouping attributes.
- if a clustered index on the grouping attribute exists, then the records are already sorted in the right way

4.9 Evaluation of a Sequence of Operators

4.9.1 Materialized Evaluation

The materialized evaluation is a naive method of evaluating a sequnce of operators. In this case, the operators are evaluated bottom up from the query plan, and each intermediate result ist saved in another file.

Disadvantages

- additional disk I/O
  - additional cost
  - additional time delay
- long response times
  - an operator cannot start producing its results before its input has been fully generated (*strictly sequential evaluation*)

4.9.2 Pipelining

The idea of *pipelining* is, that an operator could pass its result directly on to the next operator without persisting it to the disk. That means, operator evaulation starts producing results as early as possible (as soon as enough input data is available).

Different granularities are possible:

- as soon as input for next output tuple is available, or
- wait for a larger chunk of input data.

Disadvantages

- not all operators are suitable for pipelining
  - sorting
  - hash join
  - inner relation in nested loops join

4.9.3 Pipeline Realization

Pipelining can be realized by the means of an interator for each operator in a query plan.

Iterator Concept An iterator has to provide (at least) three methods:

- `open()`
  - initializes the operator evaluation (initializing data structures, allocating buffer space, ...)
- `next()`
  - generates the next output tuple, or
  - signals end of output
- `close()`
  - releases all resources

Example

- start evaluation by calling `open()` on the root operator (`sigma_q.open()`)

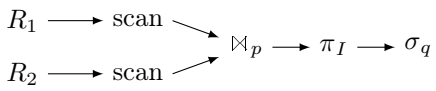


Figure 13: Query Plan Tree Example

- the `open()` call is forwarded through the plan by the operators
- when control from `sigma_q.open()` is returned to the query processor, `sigma_q.next()` is called to produce the first result tuple
- operators forward the `next()` request as needed
- as soon as the next result tuple is produced, control return to the query processor again

5 Query Optimization

The ideal optimizer

- enumerates all possible execution plans
- determines (an estimate of) the cost of each plan
- chooses the best one as the final execution plan

Problems:

- the number of possible plans may be too big → heuristics to enumerate only promising plan candidates
- the cost of each plan is only an estimate
- the time needed for query optimization may slow down the response time

5.1 Cardinality Estimation

- the value of the cost function heavily depends on the size of the intermediate results
- cardinality estimation is also importand for choosing the right buffer size

The database keeps statistics like

- number of tupels
- size of tupels
- distribution of attributes

for better cardinality estimation.

Two assumptions are typically made:

- *Independence*: values of different attributes are independent of each other
- *Uniformity*: if no information on the attribute value distribution is given, all values of an attribute have the same probability

5.1.1 Histograms

Uniform distribution is often unrealistic, therefore a histogram over the values is maintained, for a better approximation of the actual distribution.

The idea is as follows:

- divide the active domain of A into adjacent intervals (*buckets*) by choosing boundary values $b_1, b_2, b_3, \ldots$
- collect statistical information of each interval

There are two types of histograms:

- *Equi-width histograms*: all intervals have equal size
- *Equi-height histograms*: number of rows in each bucket is roughly the same

5.2 Selectivity

The selectivity is the ratio of qualifying tuples versus all tuples.

The selectivity for a selection σ with predicate P is:

$$\text{sel}_P = \frac{|\sigma_P(R)|}{|R|}$$

The selectivity of a join is:

$$\begin{aligned}\text{sel}_{R \bowtie S} &= \frac{|R \bowtie S|}{|R \times S|} \\ &= \frac{|R \bowtie S|}{|R| \cdot |S|}\end{aligned}$$

Equality on *unique attribute*:

$$\text{sel}_{R.A=c} = \frac{1}{|R|}$$

Equi-join of R and S via foreign key from R to S:

$$\text{sel}_{\bowtie_{R.A=S.B}} = \frac{1}{|S|}$$

Equality on *non-unique* attribute:

$$\text{sel}_{R.A=c} = \frac{1}{\text{NDV}(A, R)}$$

where $\text{NDV}(A, R)$ is the number of distinct valuies of attribute A in R.

Equality between attributes:

$$\text{sel}_{R.A=R.B} = \frac{1}{\max(\text{NDV}(A, R), \text{NDV}(B, R))}$$

Equi-join:

$$\text{sel}_{\bowtie_{R.A=S.B}} = \frac{1}{\max(\text{NDV}(A, R), \text{NDV}(B, S))}$$

5.2.1 Selectivity Estimates of Range Conditions

We demonstrate the selectivity estimate of range condition on this example: $\sigma_{R.A>c}(R)$

- *Case 1* without histogram: We assume that all values between the current lowest and highest values occur with the same probability.
  - for *real-valued* attribute $R.A$:

$$\text{sel}_{R.A>c} = \frac{\text{High}(A, R) - c}{\text{High}(A, R) - \text{Low}(A, R)}$$

  - for *integer-values* attribute $R.A$:

$$\text{sel}_{R.A>c} = \frac{\text{High}(A, R) - c}{\text{High}(A, R) - \text{Low}(A, R) + 1}$$

- *Case 2* with equi-depth histogram with m buckets:
  - determine i such that value c falls into bucket i
  - All buckets $i+1, \ldots, m$ entirely satisfy the condition $R.A > c$
  - inside bucket i apply the uniformity assumption as in case 1

5.2.2 Selectivity Propagation through Joins

Consider the expression $\sigma_P(R \bowtie S)$, where the join is along a foreign key from R to S and the selection predicate only uses attributes from one of R or S.

- *Case 1* P uses only attributes from R, i.e. $\sigma_P(R \bowtie S) = \sigma_p(R) \bowtie S$
  - the selection applied to R retains $\text{sel}_P \cdot |R|$ tuples of R
  - due to the foreign key, each has at most one join partner, hence $\text{sel}_P \cdot |R|$ is also an upper bound on $\sigma_P(R \bowtie S)$
- *Case 2* P uses only attributes from S, i.e. $\sigma_P(R \bowtie S) = R \bowtie \sigma_P(S)$
  - the selection applied to S retains $\text{sel}_P \cdot |S|$ tuples of S
  - by applying the uniformity assumption, we assume, that only sel_P portion of the tuples in R still have a join partner in S
  - Hence, $\text{sel}_P \cdot |R|$ is again *estimated* as an upper bound on $\sigma_P(R \bowtie S)$

5.3 Query Optimization Process

- uncorrelated subqueries are evaluated only once (not for every result from the outer query)
- subqueries can be reformulated as join
  - subqueries with `NOT IN`, `NOT EXISTS` or `<> ALL` can be unnested with an anti-join

5.4 Reducing the Search Space

For even a single block in the query, there can be a huge number of possible execution plans. The query optimizer reduces the search space by a rule-bssed optimization on the RA expressions. In particular:

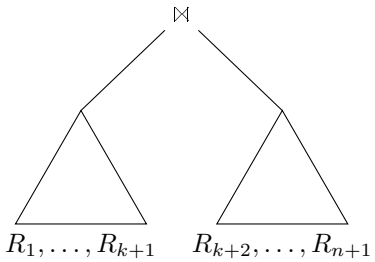
- push selections and projections down in the query tree, and
- replace cartesian products combined with selections by a join

The query optimization essentially reduces to *join optimization*, which includes join order and join implementation.

But for a join of a big number of relations the search space may still be too large to enumerate.

5.4.1 Number of Possible Combinations

A join over $n+1$ realtions requires n binary joins. Root level operator joins sub-plans of k and $n-k-1$ join operators



Let C_n be the number of possibilities to construct an ordered binary tree of n inner nodes (join operators):

$$C_n = \sum_{k=0}^{n-1} C_k \cdot C_{n-k-1}$$

The number of ordered binary trees with $n+1$ leaves is:

$$C_n = \sum_{k=0}^{n-1} C_k \cdot C_{n-k-1}$$
$$= \frac{(2n)!}{(n+1)! \cdot n!}$$

For each of these trees, we can permute the input $n+1$ relations. The number of possible join trees is thus:

$$\frac{(2n)!}{(n+1)! \cdot n!} \cdot (n+1)! = \frac{(2n)!}{n!}$$

And the combinatorial explosion gets even bigger if we take different implementations of joins into account.

Traditionally, DBMSs consider *left-deep trees* in the first place.

- significant reduction of the search space
- the inner relation is always a base relation, this allows the use of index nested loops
- is easier to implement in a pipelined fashion

5.5 Finding the Cheapest Exection Plan

- *Dynamic Programming*
  - find the cheapest plan for a join of n relations in n *passes*
  - *Pass 1*
    - consider plans for a single relation
    - for each relation find the cheapest access
  - *Pass* $k+1$
    - consider plans for joining $k+1$ relations by extending the optimal joins of k relations by one more join operation

  Dynamic programming significantly reduces the number of plans to be considered, but it is still exponential, and thus no longer feasable for a join of many tables.
- *Greedy*
  - at each step, choose the next table, such that the additional join is the cheapest
- *Heuristic*
  - restrict the search space by applying some heuristic method, e.g. generic algorithms, hill-climbing, etc.
- *PostgreSQL*
  - switches from exhaustive search to heuristic search (geqo = generic query optimizer) when the number of relations is to big
  - by default the switch to geqo happens when ≥ 12 relations are joined

5.6 Database Tuning

There are many decisions of database design:

- Which indices?
  - On which attribute or attribute combinations?
  - Which index type?
  - Clustered index?
- avoid time-critical joins
  - denormalization
  - materialized views

- avoid access to unnecessary attributes:
  - vertical partitioning of a table
- avoid access to unnecesary rows:
  - horizontal partitioning of a table (several table with same schema)

5.7 Query Tuning

- optimizers usually do not use indices for arithmetic expressions
- duplicate elimination should oinly be forced if it is really needed (avoid `DISTINCT`, replace `UNION` by `UNION ALL`)
- in case of multiple ways to formulate a join, try to use a clustered index, and try to avoid string conditions
- if views are defined via joins, are they really needed, or can the base tables be used instead
- can nested subqueries be formulated in a single `SELECT`-`FROM`-`WHERE` statement?
- compute combinations with `WITH` statement

In some DBMSs optimizer hints can be set, to influence the optimization process. This is not supported by PostgreSQL. In PostgreSQL, the user can exclude certain implementations with e.g. `SET enable_indexscan OFF`, and can change the cost of different operations.

5.8 Materialized Views

A *view* is defined by some query Q. If a view is referenced from a `FROM` clause of another query Q', then the defining query Q is expanded into query Q'.

A *materialized view* stores the result of the defining query Q and can be used directly when referenced by another query Q'.

- *Advantages*
  - time to read a materialized view is usually much smaller than time to compute the result
- *Disadvantages*
  - materialized views have to be updated whenever one of the input relations gets modified

Materialized view need to be maintained. This can happen automatically, whenever one of the input relations gets modified, preferably this is done incrementally, and not from scratch (*incremental view maintenance*), or explicitly by request from a user.

To create a materialized view in PostgreSQL, this is the command:

```
1 CREATE MATERIALIZED VIEW <name> AS <query>
```

To explicitly update the materialized view, this command is used:

```
1 REFRESH MATERIALIZED VIEW <name>
```

this recomputes the contents from scratch!

To delete a materialized view, use the `DROP` command:

```
1 DROP MATERIALIZED VIEW <name>
```