Indices

We consider B-Trees only

- key attributes: $a_1, \ldots, a_n$
- data attributes: $d_1, \ldots, d_m$
- Often: one special data attribute holding the TID of a tuple

Some notions:

- simple/complex key
- unique/non-unique index
- index-only relation (no TIDs available!)
- clustered/non-clustered index
Clustered vs. Non-Clustered B-Tree

- clustering is not always possible (or even desireable)
Single Index Access Path - Point Query

Exact match query:

```
select name
from Emp
where eno = 1077
```

Translation:

\[
\Pi_{\text{name}}(\chi_e:*.tid,\text{name}:e.name(Emp_{eno}[x; eno = 1077]))
\]

Alternative translation using d-join:

\[
\Pi_{\text{name}}(Emp_{eno}[x; eno = 1077] \bowtie \chi_e:*.tid,\text{name}:e.name(\Box))
\]

(x: holds ptr to index entry; *: dereference TID, \(\Box\) is a singleton scan)
Single Index Access Path - Range Query

Range query:

```sql
select name
from Emp
where age ≥ 25 and age ≤ 35
```

Translation:

```
Π_{name}(χ_{e:*}, name: e.name(Emp_{age}[x; 25 ≤ age; age ≤ 35]))
```

(Start and Stop condition)
Single Index Access Path - Sequential I/O

Turning random I/O into sequential I/O:

\[ \Pi_{\text{name}}(\chi_{\text{e:*tid, name:e.name}}(\text{sort}_{x.tid}(\text{Emp}_{age}[x; 25 \leq \text{age}; \text{age} \leq 35; \text{tid}]))) \]

Note: explicit projection the TID attribute of the index within the index scan.
Single Index Access Path - Sorted Output

Query demanding ordered output:

```sql
select name, age
from Emp
where age ≥ 25 and age ≤ 35
order by age
```

Translation:

```
Π_{name,age}(\chi_{e:*x.tid,name:e.name}(Emp_{age[x;25 ≤ age;age ≤ 35]}))
```

Note: output of index scan ordered on its key attributes
This order can be exploited in many ways: e.g.: subsequent merge join
Single Index Access Path - Sorted Output (2)

Turning random I/O into sequential I/O requires resort:

\[ \Pi_{name, age}(sort_{age}(\chi_{e:*tid, name:e.name}(sort_{tid}(Emp_{age}[x; 25 \leq age; age \leq 35; tid])))]) \]

Possible speedup of sort by dense numbering:

\[ \Pi_{name, age}(sort_{rank}(\chi_{e:*tid, name:e.name}(sort_{tid}(\chi_{rank:counter++}(Emp_{age}[x; 25 \leq age; age \leq 35; tid])))))) \]
Single Index Access Path - Other Predicates

Some predicates not index sargable but still useful as residual predicates:

```sql
select name
from Emp
where age ≥ 25 and age ≤ 35 and age ≠ 30
```

Translation:

$$\Pi_{\text{name}}(\chi_{e:*x.tid,name:e.name}(\text{Emp}_{\text{age}[x; 25 \leq \text{age}; \text{age} \leq 35; \text{age} \neq 30])))$$
Single Index Access Path - Other Predicates (2)

Non-inclusive bounds:

```
select name
from Emp
where age > 25 and age < 35
```

If supported by index:

```
Π_{name}(χ_{e:*x.tid,name:e.name}(Emp_{age}[x; 25 < age; age < 35]))
```

If unsupported:

```
Π_{name}(χ_{e:*x.tid,name:e.name}(Emp_{age}[x; 25 ≤ age; age ≤ 35; age ≠ 25, age ≠ 35]))
```

Especially for predicates on strings this might be expensive.
Single Index Access Path - Ranges

Start and stop conditions are optional:

```sql
select name
from Emp
where age ≥ 60
```

or

```sql
select name
from Emp
where age ≤ 20
```
Single Index Access Path - No Range

Full index scan also useful:

```sql
select count(*)
from Emp
```

Also works for sum/avg.
(notion: index only query)
Single Index Access Path - No Range (2)

Min/max even more efficient:

```
select min/max(salary)
from Emp
```
Single Index Access Path - No Range (3)

```sql
select name
from Emp
where salary = (select max(salary)
    from Emp)
```

Alternatives: one or two descents into the index.
Single Index Access Path - No Range (4)

Full index scan:

```sql
select salary
from Emp
order by salary
```

Translation:

\[ Emp_{\text{salary}} \]
Single Index Access Path - String Ranges

Predicate on string attribute:

\[
\text{select name, salary} \\
\text{from Emp} \\
\text{where name} \geq 'Maaa' \\
\]

Start condition: \( 'Maaa' \leq name \)

\[
\text{select name, salary} \\
\text{from Emp} \\
\text{where name like 'M\%'} \\
\]

Start condition: \( 'M' \leq name \)
**Single Index Access Path**

- an *access path* is a plan fragment with building blocks concerning a single database items.
- hence, every building block is an access path.
- above plans mostly touch two database items: a relation and an index on some attribute of that relation.
- if we say that an index concerns the relation that it indexes, such a fragment is an access path.
- for relational systems, the most general case of an access path uses several indices to retrieve the tuples of a single relation.
- we will see examples of these more complex access paths in the following section.
- a query that can be answered solely by accessing indexes is called an *index only query*. 
Single Index Access Path - Complex Predicates

Query with IN:

```sql
select name
from Emp
where age in {28, 29, 31, 32}
```

Take min/max value for start/stop key plus one of the following as the residual predicate:

- $age = 28 \lor age = 29 \lor age = 31 \lor age = 32$
- $age \neq 30$
Single Index Access Path - Complex Predicates (2)

A case for the d-join:

```sql
select name
from Emp
where salary in {1111, 11111, 111111}
```

With $Sal = \{[s: 1111], [s: 11111], [s: 111111]\}$:

$$Sal[S] \land \chi_{e:*tid,name:e.name}(Emp_{\text{salary}[x; salary = S.s; tid]})$$

- gap skipping/zig-zag skipping
Single Index Access Path - Compound Keys

In general an index can have a complex key comprising of key attributes $k_1, \ldots, k_n$ and data attributes $d_1, \ldots, d_m$. Besides a full index scan, the index can be descended to directly search for the desired tuple(s):

If the search predicate is of the form

$$k_1 = c_1 \land k_2 = c_2 \land \ldots \land k_j = c_j$$

for some constants $c_i$ and some $j \leq n$, we can generate the start and stop condition

$$k_1 = c_1 \land \ldots \land k_j = c_j.$$
With ranges things become more complex and highly dependent on the implementation of the facilities of the B-Tree:

\[ k_1 = c_1 \land k_2 \geq c_2 \land k_3 = c_3 \]

Obviously, we can generate the start condition \( k_1 = c_1 \land k_2 \geq c_2 \) and the stop condition \( k_1 = c_1 \).

Here, we neglected the condition on \( k_3 \) which becomes a residual predicate. However, with some care we can extend the start condition to \( k_1 = c_1 \land k_2 \geq c_2 \land k_3 = c_3 \):

we only have to keep \( k_3 = c_3 \) as a residual predicate since for \( k_2 \) values larger than \( c_2 \) values different from \( c_3 \) can occur for \( k_3 \).
Single Index Access Path - Compound Keys (2)

If closed ranges are specified for a prefix of the key attributes as in

\[ a_1 \leq k_1 \leq b_1 \land \ldots \land a_j \leq k_j \leq b_j \]

we can generate the start key \( k_1 = a_1 \land \ldots \land k_j = a_j \), the stop key \( k_1 = b_1 \land \ldots \land k_j = b_j \), and

\[ a_2 \leq k_2 \leq b_2 \land \ldots \land a_j \leq k_j \leq b_j \]

as the residual predicate.

If for some search key attribute \( k_j \) the lower bound \( a_j \) is not specified, the start condition can not contain \( k_j \) and any \( k_{j+i} \).

If for some search key attribute \( k_j \) the upper bound \( b_j \) is not specified, the stop condition can not contain \( k_j \) and any \( k_{j+i} \).
Single Index Access Path - Improvements

Two further enhancements of the B-Tree functionality possibly allow for alternative start/stop conditions:

- The B-Tree implementatio allows to specify the order (ascending or descending) for each key attribute individually.
- The B-Tree implementation implements forward and backward scans.
Single Index Access Path - Improvements (2)

Consider search predicate:

```
haircolor = 'blond' and height between 180 and 190
```

and index on

```
sex, haircolor, height
```

There are only the two values male and female available for sex. Rewrite:

```
(sex = 'm' and haircolor = 'blond' and height between 180 and 190) or (sex = 'f' and haircolor = 'blond' and height between 180 and 190)
```

Improvement: determine rewrite at query execution time in conjunction with gap skipping.
Multi Index Access Path - Example

Query:

```sql
select * 
from Camera 
where megapixel > 5 and distortion < 0.05 
and noise < 0.01 
zoomMin < 35 and zoomMax > 105
```

Indexes on all attributes
Multi Index Access Path - Example (2)

Translation:

((((
    Camera\text{megapixel}[c; \text{megapixel} > 5; tid]
    \cap
    Camera\text{distortion}[c; \text{distortion} < 0.05; tid])
    \cap
    Camera\text{noise}[c; \text{noise} < 0.01; tid])
    \cap
    Camera\text{zoomMin}[c; \text{zoomMin} < 35; tid])
    \cap
    Camera\text{zoomMax}[c; \text{zoomMax} > 105; tid])

Then dereference

- Notion: index and-ing/and merge (bitmap index)
Multi Index Access Path - Combining

Questions:

• In which order do we intersect the TID sets resulting from the index scans?

• Do we really apply all indexes before dereferencing the TIDs?

The answer to the latter question is clearly “no”, if the next index scan is more expensive than accessing the records in the current TID list. It can be shown that the indexes in the cascade of intersections are ordered on increasing $(f_i - 1)/c_i$ terms where $f_i$ is the selectivity of the index and $c_i$ its access cost. Further, we can stop as soon as accessing the original tuples in the base relation becomes cheaper than intersecting with another index and subsequently accessing the base relation.
Multi Index Access Path - Combining (2)

Index-oring (or merge):

```sql
select  *
from    Emp
where   yearsOfEmployment ≥ 30
       or age ≥ 65
```

Translation:

```
Emp\_{yearsOfEmployment}[c; yearsOfEmployment ≥ 30; tid] \cup Emp\_{age}[c; age ≥ 65; tid]
```

Attention: duplicates
Optimal translation of complex boolean expressions? Factorization?
Multi Index Access Path - Combining (3)

Index differencing:

```sql
select  *  
from    Emp  
where   yearsOfEmployment \neq 10  
        and  age \geq  65
```

Translation:

```
Emp_{age}[c; age \geq 65; tid] \setminus Emp_{yearsOfEmployment}[c; yearsOfEmployment = 10; tid]
```
Multi Index Access Path - Combining (3)

Non-restrictive index sargable predicates (more than half of the index has to be read):

```sql
select * 
from Emp 
where yearsOfEmployment ≤ 5 
    and age ≤ 60
```

Then

\[ Emp_{yearsOfEmployment}[c; yearsOfEmployment ≤ 5; tid] \setminus Emp_{age}[c; age > 60; tid] \]

could be more efficient than

\[ Emp_{yearsOfEmployment}[c; yearsOfEmployment ≤ 5; tid] \cap Emp_{age}[c; age ≤ 60; tid] \]
Indices and Join

1. speed up joins by index exploitation
2. make join a general index processing operation

(intersection is similar to join (for sets))
Indices and Join (2)

Turn map

$$\chi_{e:*tid, name:e.name}(Emp_{salary}[x; 25 \leq age \leq 35; tid])$$

into d-join

$$Emp_{salary}[x; 25 \leq age \leq 35; tid] \bowtie \chi_{e:*tid, name:e.name}(\square)$$

or even join

$$Emp_{salary}[x; 25 \leq age \leq 35] \bowtie_x tid = e.tid Emp[e]$$

Variants: sorting at different places (by plan generator)

• pro: flexibility
• contra: large search space
Indices and Join (3)

Query:

```sql
select name, age
from Person
where name like 'R%' and age between 40 and 50
```

Translation:

```
Π_{name,age}(  
    Emp_{age}[a; 40 \leq age \leq 50; TIDA, age] 
    \land TIDA = TIDn 
    Emp_{name}[n; name \geq 'R'; name < 'S'; TIDn, name] )
```
The query

\[
\text{select } * \\
\text{from } \quad \text{Emp } e, \text{ Dept } d \\
\text{where } \quad e.\text{name} = 'Maier' \text{ and } e.\text{dno} = d.\text{dno}
\]

can be directly translated to

\[
\sigma_{e.\text{name}='Maier'}(\text{Emp}[e]) \bowtie_{e.\text{dno}=d.\text{dno}} \text{Dept}[d]
\]
Indices and Join (5)

If there are indexes on `Emp.name` and `Dept.dno`, we can replace
\[ \sigma_{e.name='Maier'}(Emp[e]) \]
by an index scan as we have seen previously:
\[ \chi_{e:*x.tid(Emp_{name}[x; name='Maier'])} \]
Indices and Join (6)

With a d-join:

$$Emp_{name}[x; \text{name} = "Maier"] \bowtie_{\chi_e:*x.tid(\square)}$$

Abbreviate $Emp_{name}[x; \text{name} = "Maier"]$ by $E_i$
Abbreviate $\chi_e:*x.tid(\square)$ by $E_a$. 
Indices and Join (7)

Use index on Dept.dno:

\[ E_i \Join E_a \Join \text{Dept}_{dno}[y; y.dno = dno] \]

Dereference TIDs (index nested loop join):

\[ E_i \Join E_a \Join \text{Dept}_{dno}[y; y.dno = dno; \text{dtid} : y.tid] \Join \chi_{u:*\text{dtid}}(\Box) \]

Abbreviate \( \text{Dept}_{dno}[y; y.dno = dno; \text{dtid} : y.tid] \) by \( D_i \)

Abbreviate \( \chi_{u:*\text{dtid}}(\Box) \) by \( D_a \)

Fully abbreviated, the expression then becomes

\[ E_i \Join E_a \Join D_i \Join D_a \]
Indices and Join - Performance Improvements

Optimizations: sorting the outer of a d-join is useful under several circumstances since it may

- turn random I/O into sequential I/O and/or
- avoid reading the same page twice.

In our example expression:
Indices and Join - Performance Improvements (2)

- We can sort the result of expression $E_i$ on TID in order to turn random I/O into sequential I/O, if there are many employees named ”Maier”.
- We can sort the result of the expression $E_i \bowtie E_a$ on dno for two reasons:
  - If there are duplicates for dno, i.e. there are many employees named ”Maier” in each department, then this guarantees that no index page (of the index Dept.dno) has to be read more than once.
  - If additionally Dept.dno is a clustered index or Dept is an index-only table contained in Dept.dno then large parts of the random I/O can be turned into sequential I/O.
  - If the result of the inner is materialized (see below), then only one result needs to be stored. Note that sorting is not necessary but grouping would suffice to avoid duplicate work.
- We can sort the result of the expression $E_i \bowtie E_a \bowtie D_i$ on dtid for the same reasons as mentioned above for sorting the result of $E_i$ on TID.
Indices and Join - Temping the Inner

Typically, many employees will work in a single department and possibly several of them are called "Maier". For everyone of them, we can be sure that there exists at most one department.

Let us assume that referential integrity has been specified. Then there exists exactly one department for every employee. We have to find a way to rewrite the expression

\[ E_i \bowtie E_a \bowtie \text{Dept}_{dno}[y; y.dno = dno; dtid : y.\text{rid}] \]

such that the mapping \( dno \rightarrow dtid \) is explicitly materialized (or, as one could also say, cached).
Indices and Join - Temping the Inner (2)

Use $\chi^{mat}$:

$$E_i \bowtie E_a \bowtie \chi^{mat}_{tid:(\text{Dept}_{dno}[y; y.dno=dno]).tid()}$$
If we further assume that the outer \((E_i \bowtie E_a)\) is sorted on dno, then it suffices to remember only the TID for the latest dno. We define the map operator \(\chi^{mat,1}\) to do exactly this. A more efficient plan could thus be

\[
\text{sort}_{dno}(E_i \bowtie E_a) \bowtie \chi^{mat,1}_{dtid}(\text{Dept}_{dno}[y; y.dno = dno]).tid(\square)
\]

where, strictly speaking, sorting is not necessary: grouping would suffice.
Indices and Join - Temping the Inner (4)

Consider: $e_1 \bowtie e_2$

The free variables used in $e_2$ must be a subset of the variables (attributes) produced by $e_1$, i.e. $\mathcal{F}(e_2) \subseteq \mathcal{A}(e_1)$.

Even if $e_1$ does not contain duplicates, the projection of $e_1$ on $\mathcal{F}(e_2)$ may contain duplicates.

If so, materialization could pay off.

However, in general, for every binding of the variables $\mathcal{F}(e_2)$, the expression $e_2$ may produce several tuples.

This means that using $\chi^{mat}$ is not sufficient.
Indices and Join - Temping the Inner (5)

The query

\[
\text{select } \ast \\
\text{from } \text{Emp } e, \text{Wine } w \\
\text{where } e.\text{yearOfBirth } = w.\text{year}
\]

has the usual suspects as plans.
Assume we have only wines from a few years.
Then, it might make sense to consider the following alternative:

\[
\text{Wine}[w] \bowtie_{e.\text{yearOfBirth } = w.\text{year}}(\text{Emp}[e])
\]

Problem: scan \text{Emp} once for each \text{Wine} tuple
Duplicates in \text{Wine}.\text{year}: scan \text{Emp} only once per \text{Wine}.\text{year} value
Indices and Join - Temping the Inner (6)

The `memox` operator performs caching:

\[ Wine[w] \bowtie_{\text{memox}} (\sigma_{e.\text{yearOfBirth} = w.\text{year}}(\text{Emp}[e])) \]

Sorting still beneficial:

\[ \text{sort}_{w.\text{year}} (Wine[w]) \bowtie_{\text{memox}^1} (\sigma_{e.\text{yearOfBirth} = w.\text{year}}(\text{Emp}[e])) \]
Indices and Join - Temping the Inner (7)

Things can become even more efficient if there is an index on Emp.yearOfBirth:

\[
\text{sort}_{w.\text{year}}(Wine[w]) \quad \text{mempx}^1(Emp_{\text{yearOfBirth}}[x; x.\text{yearOfBirth} = w.\text{year}] \quad \text{mempx}^1(Emp_{\text{yearOfBirth}}[x; x.\text{yearOfBirth} = w.\text{year}] 
\]
Indices and Join - Temping the Inner (8)

Indexes on Emp.yearOfBirth and Wine.year.
Join result of index scans.
Since the index scan produces its output ordered on the key attributes, a simple merge join suffices (and we are back at the latter):

\[ \text{Emp}_{\text{yearOfBirth}}[x] \times_{x.\text{yearOfBirth}=y.\text{year}} \text{Wine}_{\text{year}}[y] \]
Remarks on Access Path Generation

Side-ways information passing
Consider $R \bowtie_{R.a = S.b} S$

- min/max for restriction on other join argument
- full projection on join attributes (leads to semi-join)
- bitmap representation of the projection