Indices

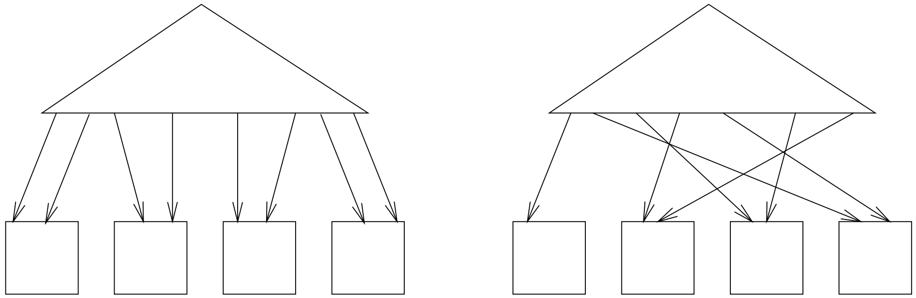
We consider B-Trees only

- 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:

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

Alternative translation using d-join:

$$\Pi_{name}(Emp_{eno}[x; eno = 1077] \ltimes \chi_{e:*.tid,name:e.name}(\Box))$$

(x: holds ptr to index entry; *: dereference TID, $\Box$ is a singleton scan)

Single Index Access Path - Range Query

Range query:

select name
from Emp
where age $\geq$ 25 **and** age $\leq$ 35

Translation:

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

(Start and Stop condition)

Single Index Access Path - Sequential I/O

Turning random I/O into sequential I/O:

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

Note: explicit projection the TID attribute of the index within the index scan.

Single Index Access Path - Sorted Output

Query demanding ordered output:

select	name, age
from	Emp
where	age $\geq$ 25 **and** age $\leq$ 35
order by	age

Translation:

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

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:

$$\begin{array}{l}
\Pi_{name,age}(\\
\quad sort_{rank}(\\
\quad\quad \chi_{e:*tid,name:e.name}(\\
\quad\quad\quad sort_{tid}(\\
\quad\quad\quad\quad \chi_{rank:counter++}(\\
\quad\quad\quad\quad\quad Emp_{age}[x; 25 \leq age; age \leq 35; tid])))))
\end{array}$$

Single Index Access Path - Other Predicates

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

select name
from Emp
where age $\geq$ 25 **and** age $\leq$ 35 **and** age $\neq$ 30

Translation:

$$\Pi_{name}(\chi_{e:*x.tid,name:e.name}(Emp_{age}[x; 25 \leq age; age \leq 35; age \neq 30]))$$

Single Index Access Path - Other Predicates (2)

Non-inclusive bounds:

select name
from Emp
where age $>$ 25 **and** age $<$ 35

If supported by index:

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

If unsupported:

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

Especially for predicates on strings this might be expensive.

Single Index Access Path - Ranges

Start and stop conditions are optional:

select name
from Emp
where age $\geq$ 60

or

select name
from Emp
where age $\leq$ 20

Single Index Access Path - No Range

Full index scan also useful:

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)

```
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:

select salary
from Emp
order by salary

Translation:

$$Emp_{salary}$$

Single Index Access Path - String Ranges

Predicate on string attribute:

select name, salary
from Emp
where name $\geq$ 'Maaa'

Start condition: $'Maaa' \leq name$

select name, salary
from Emp
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`:

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:

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

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

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

- 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

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

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

$$k_1 = c_1 \wedge \ldots \wedge k_j = c_j.$$

Single Index Access Path - Compound Keys

With ranges things become more complex and highly dependent on the implementation of the facilities of the B-Tree:

$$k_1 = c_1 \wedge k_2 \geq c_2 \wedge k_3 = c_3$$

Obviously, we can generate the start condition $k_1 = c_1 \wedge k_2 \geq c_2$ 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
$k_1 = c_1 \wedge k_2 \geq c_2 \wedge k_3 = c_3$:
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

$$a_1 \leq k_1 \leq b_1 \wedge \ldots \wedge a_j \leq k_j \leq b_j$$

we can generate the start key $k_1 = a_1 \wedge \ldots \wedge k_j = a_j$, the stop key $k_1 = b_1 \wedge \ldots \wedge k_j = b_j$, and

$$a_2 \leq k_2 \leq b_2 \wedge \ldots \wedge a_j \leq k_j \leq b_j$$

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 implemenation 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:

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:

$$
\begin{aligned}
&((((\\
&\quad Camera_{megapixel}[c; megapixel > 5; tid]\\
&\quad \cap\\
&\qquad Camera_{distortion}[c; distortion < 0.05; tid])\\
&\qquad \cap\\
&\qquad\quad Camera_{noise}[c; noise < 0.01; tid])\\
&\qquad\quad \cap\\
&\qquad\qquad Camera_{zoomMin}[c; zoomMin < 35; tid])\\
&\qquad\qquad \cap\\
&\qquad\qquad\quad Camera_{zoomMax}[c; zoomMax > 105; tid])
\end{aligned}
$$

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):

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

Translation:

$Emp_{yearsOfEmployment}[c; yearsOfEmployment \geq 30; tid] \cup Emp_{age}[c; age \geq 65; ti$

Attention: duplicates
Optimal translation of complex boolean expressions? Factorization?

Multi Index Access Path - Combining (3)

Index differencing:

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

Translation:

$Emp_{age}[c; age \geq 65; tid] \setminus Emp_{yearsOfEmployment}[c; yearsOfEmployment = 10; ti$

Multi Index Access Path - Combining (3)

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

select *
from Emp
where yearsOfEmployment $\leq$ 5
and age $\leq$ 60

Then

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

could be more efficient than

$Emp_{yearsOfEmployment}[c; yearsOfEmployment \leq 5; tid] \cap Emp_{age}[c; age \leq 60; tid$

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

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

or even join

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

Variants: sorting at different places (by plan generator)

- pro: flexibility
- contra: large search space

Indices and Join (3)

Query:

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

Translation:

$$
\begin{array}{l}
\Pi_{name,age}(\\
\quad Emp_{age}[a; 40 \leq age \leq 50; TIDa, age] \\
\quad\quad \bowtie_{TIDa=TIDn} \\
\quad Emp_{name}[n; name \geq' R'; name <' S'; TIDn, name])
\end{array}
$$

Indices and Join (4)

The query

select $*$
from *Emp e, Dept d*
where *e.name = 'Maier' and e.dno = d.dno*

can be directly translated to

$$\sigma_{e.name=''Maier''}(Emp[e]) \bowtie_{e.dno=d.dno} Dept[d]$$

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:

$$Emp_{name}[x; name ='' \ Maier''] \ltimes \chi_{e:*x.tid}(\Box)$$

Abbreviate $Emp_{name}[x; name ='' \ Maier'']$ by E_i
Abbreviate $\chi_{e:*x.tid}(\Box)$ by E_a.

Indices and Join (7)

Use index on `Dept.dno`:

$$E_i \bowtie E_a \bowtie Dept_{dno}[y; y.dno = dno]$$

Dereference TIDs (*index nested loop join*):

$$E_i \bowtie E_a \bowtie Dept_{dno}[y; y.dno = dno; dtid : y.tid] \bowtie \chi_{u:*dtid}(\square)$$

Abbreviate $Dept_{dno}[y; y.dno = dno; dtid : y.tid]$ by D_i
Abbreviate $\chi_{u:*dtid}(\square)$ by D_a
Fully abbreviated, the expression then becomes

$$E_i \bowtie E_a \bowtie D_i \bowtie D_a$$

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 intregrity has been specified.
Then there exists exactly one department for every employee.
We have to find a way to rewrite the expression

$$E_i \bowtie E_a \bowtie Dept_{dno}[y; y.dno = dno; dtid : y.rid]$$

such that the mapping $dno \longrightarrow dtid$ is explicitly materialized (or, as one could also say, *cached*).

Indices and Join - Temping the Inner (2)

Use χ^{mat}:

$$E_i \bowtie E_a \bowtie \chi^{mat}_{tid:(Dept_{dno}[y;y.dno=dno]).tid}(\Box)$$

Indices and Join - Temping the Inner (3)

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

$$sort_{dno}(E_i \bowtie E_a) \bowtie \chi^{mat,1}_{dtid:(Dept_{dno}[y;y.dno=dno]).tid}(\square)$$

where, strictly speaking, sorting is not necessary: grouping would suffice.

Indices and Join - Temping the Inner (4)

Consider: $e_1 \ltimes e_2$
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 χ^{mat} is not sufficient.

Indices and Join - Temping the Inner (5)

The query

select *
from Emp e, Wine w
where e.yearOfBirth = w.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:

$$Wine[w] \ltimes \sigma_{e.yearOfBirth=w.year}(Emp[e])$$

Problem: scan `Emp` once for each `Wine` tuple
Duplicates in `Wine.year`: scan `Emp` only once per `Wine.year` value

Indices and Join - Temping the Inner (6)

The memox operator performs caching:

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

Sorting still beneficial:

$$sort_{w.year}(Wine[w]) \ltimes memox^{1}(\sigma_{e.yearOfBirth=w.year}(Emp[e]))$$

Indices and Join - Temping the Inner (7)

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

$$sort_{w.year}(Wine[w])$$
$$\Join memox^1(Emp_{yearOfBirth}[x; x.yearOfBirth = w.year] \Join \chi_{e:*(x.tid)}(\square))$$

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):

$$Emp_{yearOfBirth}[x] \bowtie^{merge}_{x.yearOfBirth=y.year} Wine_{year}[y]$$

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