Database Systems Architecture Q & A

Stijn Vansummeren

Translation of SQL into the relational algebra

In the examples that follow, we will use the following database:

- Movie(title: `string`, year: `int`, length: `int`, genre: `string`, studioName: `string`, producerC#: `int`)
- MovieStar(name: `string`, address: `string`, gender: `char`, birthdate: `date`)
- StarsIn(movieTitle: `string`, movieYear: `string`, starName: `string`)
- MovieExec(name: `string`, address: `string`, CERT#: `int`, netWorth: `int`)
- Studio(name: `string`, address: `string`, presC#: `int`)

Translation of SQL into the relational algebra

Select-from-where statements without subqueries

SQL:

```
SELECT movieTitle
FROM StarsIn S, MovieStar M
WHERE S.starName = M.name AND M.birthdate = 1960
```

Algebra:

- Translate From-clause into cartesian product (ρ, $\times$)
- Translate Where-clause into selection (σ)
- Translate Select-clause into projection (π)

Translation of SQL into the relational algebra

Correlated subqueries

A subquery can refer to attributes of relations that are introduced in an outer query.

```
SELECT movieTitle
FROM StarsIn S
WHERE EXISTS (SELECT name
              FROM MovieStar
              WHERE birthdate=1960 AND name=S.starName)
```

Definition

- We call such subqueries correlated subqueries.
- The "outer" relations from which the correlated subquery uses some attributes are called the context relations of the subquery.
- The set of all attributes of all context relations of a subquery are called the parameters of the subquery.

Translation of SQL into the relational algebra

Translation of correlated select-from-where subqueries

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND EXISTS (SELECT name
            FROM MovieStar
            WHERE birthdate=1960 AND name= S.starName)
```

Translation of SQL into the relational algebra

Translation of correlated select-from-where subqueries

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND EXISTS (SELECT name
            FROM MovieStar
            WHERE birthdate=1960 AND name= S.starName)
```

1. We first translate the EXISTS subquery.

$$\pi_{\texttt{name}}\, \sigma_{\substack{\texttt{birthdate}=1960 \\ \wedge \texttt{name}=\texttt{S.starName}}} (\texttt{MovieStar})$$

Translation of SQL into the relational algebra

Translation of correlated select-from-where subqueries

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND EXISTS (SELECT name
            FROM MovieStar
            WHERE birthdate=1960 AND name= S.starName)
```

1. We first translate the EXISTS subquery.

$$\pi_{\text{name}}\, \sigma_{\substack{\text{birthdate}=1960 \\ \wedge \text{name}=\text{S.starName}}} (\text{MovieStar})$$

Since we are translating a correlated subquery, however, we need to add the context relations and parameters for this translation to make sense.

$$\pi_{\text{S.movieTitle},\text{S.movieYear},\text{S.starName},\text{name}}\, \sigma_{\substack{\text{birthdate}=1960 \\ \wedge \text{name}=\text{S.starName}}} (\text{MovieStar} \times \rho_S(\text{StarsIn}))$$

Translation of SQL into the relational algebra

Translation of correlated select-from-where subqueries

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND EXISTS (SELECT name
            FROM MovieStar
            WHERE birthdate=1960 AND name= S.starName)
```

2. Next, we translate the FROM clause of the outer query. This gives us:

$$\boldsymbol{\rho}_S(\texttt{StarsIn}) \times \boldsymbol{\rho}_M(\texttt{Movie})$$

Translation of SQL into the relational algebra

Translation of correlated select-from-where subqueries

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND EXISTS (SELECT name
            FROM MovieStar
            WHERE birthdate=1960 AND name= S.starName)
```

3. We "synchronize" these subresults by means of a join. From the subquery we only need to retain the parameter attributes.

$$(\boldsymbol{\rho}_S(\texttt{StarsIn}) \times \boldsymbol{\rho}_M(\texttt{Movie})) \bowtie \boldsymbol{\pi}_{\texttt{S.movieTitle,S.movieYear,S.starName}} \boldsymbol{\sigma}_{\substack{\texttt{birthdate}=1960 \\ \wedge \texttt{name}=\texttt{S.starName}}} (\texttt{MovieStar} \times \boldsymbol{\rho}_S(\texttt{StarsIn}))$$

Translation of SQL into the relational algebra

Translation of correlated select-from-where subqueries

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND EXISTS (SELECT name
            FROM MovieStar
            WHERE birthdate=1960 AND name= S.starName)
```

4. We can simplify this by omitting the first $\rho_S(\texttt{StarsIn})$

$$\rho_M(\texttt{Movie}) \bowtie \pi_{\texttt{S.movieTitle,S.movieYear,S.starName}} \sigma_{\substack{\texttt{birthdate}=1960 \\ \wedge \texttt{name}=\texttt{S.starName}}} (\texttt{MovieStar} \times \rho_S(\texttt{StarsIn}))$$

Translation of SQL into the relational algebra

Translation of correlated select-from-where subqueries

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND EXISTS (SELECT name
            FROM MovieStar
            WHERE birthdate=1960 AND name= S.starName)
```

5. Finally, we translate the remaining subquery-free conditions in the WHERE clause, as well as the SELECT list

$$\pi_{\texttt{S.movieTitle,M.studioName}}\, \sigma_{\texttt{S.movieYear}>=2000 \wedge \texttt{S.movieTitle}=\texttt{M.title}} \Big(\rho_M(\texttt{Movie}) \bowtie \pi_{\texttt{S.movieTitle,S.movieYear,S.starName}} \sigma_{\substack{\texttt{birthdate}=1960 \\ \wedge \texttt{name}=\texttt{S.starName}}} \big(\texttt{MovieStar} \times \rho_S(\texttt{StarsIn})\big)\Big)$$

Translation of SQL into the relational algebra

Translation of correlated select-from-where subqueries

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND NOT EXISTS (SELECT name
                FROM MovieStar
                WHERE birthdate=1960 AND name= S.starName)
```

Translation of SQL into the relational algebra

Translation of correlated select-from-where subqueries

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND NOT EXISTS (SELECT name
                FROM MovieStar
                WHERE birthdate=1960 AND name= S.starName)
```

1. We first translate the NOT EXISTS subquery.

$$\pi_{\text{name}}\, \sigma_{\substack{\text{birthdate}=1960 \\ \wedge \text{name}=\text{S.starName}}} (\text{MovieStar})$$

Translation of SQL into the relational algebra

Translation of correlated select-from-where subqueries

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND NOT EXISTS (SELECT name
                FROM MovieStar
                WHERE birthdate=1960 AND name= S.starName)
```

1. We first translate the NOT EXISTS subquery.

$$\pi_{\text{name}}\, \sigma_{\substack{\text{birthdate}=1960 \\ \wedge \text{name}=\text{S.starName}}} (\text{MovieStar})$$

Since we are translating a correlated subquery, however, we need to add the context relations and parameters for this translation to make sense.

$$\pi_{\text{S.movieTitle},\text{S.movieYear},\text{S.starName},\text{name}}\, \sigma_{\substack{\text{birthdate}=1960 \\ \wedge \text{name}=\text{S.starName}}} (\text{MovieStar} \times \rho_S(\text{StarsIn}))$$

Translation of SQL into the relational algebra

Translation of correlated select-from-where subqueries

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND NOT EXISTS (SELECT name
                FROM MovieStar
                WHERE birthdate=1960 AND name= S.starName)
```

2. Next, we translate the FROM clause of the outer query. This gives us:

$$\boldsymbol{\rho}_S(\texttt{StarsIn}) \times \boldsymbol{\rho}_M(\texttt{Movie})$$

Translation of SQL into the relational algebra

Translation of correlated select-from-where subqueries

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND NOT EXISTS (SELECT name
                FROM MovieStar
                WHERE birthdate=1960 AND name= S.starName)
```

3. We then "synchronize" these subresults by means of an antijoin. From the subquery we only need to retain the parameter attributes.

$$(\boldsymbol{\rho}_S(\texttt{StarsIn}) \times \boldsymbol{\rho}_M(\texttt{Movie})) \bowtie$$
$$\boldsymbol{\pi}_{\texttt{S.movieTitle,S.movieYear,S.starName}} \boldsymbol{\sigma}_{\substack{\texttt{birthdate}=1960 \\ \wedge \texttt{name}=\texttt{S.starName}}} (\texttt{MovieStar} \times \boldsymbol{\rho}_S(\texttt{StarsIn}))$$

Here, the antijoin $R \bowtie S \equiv R - (R \bowtie S)$.

Simplification is not possible: we cannot remove the first $\boldsymbol{\rho}_S(\texttt{StarsIn})$.

Translation of SQL into the relational algebra

Translation of correlated select-from-where subqueries

```
SELECT S.movieTitle, M.studioName
FROM StarsIn S, Movie M
WHERE S.movieYear >= 2000
AND S.movieTitle = M.title
AND NOT EXISTS (SELECT name
                FROM MovieStar
                WHERE birthdate=1960 AND name= S.starName)
```

4. Finally, we translate the remaining subquery-free conditions in the `WHERE` clause, as well as the `SELECT` list

$$\pi_{\texttt{S.movieTitle,M.studioName}}\, \sigma_{\texttt{S.movieYear}>=2000 \wedge \texttt{S.movieTitle}=\texttt{M.title}} \Big(\big(\rho_S(\texttt{StarsIn}) \times \rho_M(\texttt{Movie}) \big) \bowtie \pi_{\texttt{S.movieTitle,S.movieYear,S.starName}} \sigma_{\substack{\texttt{birthdate}=1960 \\ \wedge \texttt{name}=\texttt{S.starName}}} \big(\texttt{MovieStar} \times \rho_S(\texttt{StarsIn}) \big) \Big)$$

Translation of SQL into the relational algebra

GROUP BY and HAVING

SQL:

```
SELECT name, SUM(length)
FROM MovieExec, Movie
WHERE cert# = producerC#
GROUP BY name
HAVING MIN(year) < 1930
```

Algebra:

$$\pi_{\texttt{name,SUM(length)}}\, \sigma_{\texttt{MIN(year)}<1930}\, \gamma_{\texttt{name,MIN(year),SUM(length)}}\, \sigma_{\texttt{cert\#=producerC\#}}(\texttt{MovieExec} \times \texttt{Movie})$$

Translation of SQL into the relational algebra

Task

Translate the following SQL-query into an expression of the relational algebra.

```
SELECT MAX(P.price), S.sname
FROM Parts P, Suppliers S
WHERE S.city = Ham
AND (P.Price, S.city) IN
(SELECT P2.Price, S2.city FROM Parts P2, Supply Y, Suppliers S2
 WHERE P2.pid = Y.sid AND Y.pid = S2.pid
 AND S.sid = S2.sid AND P.pid = P2.pid)
GROUP BY S.sname
```

Translation of SQL into the relational algebra

Solution

Normalization gives

```
SELECT MAX(P.price), S.sname
FROM Parts P, Suppliers S
WHERE S.city = Ham
AND EXISTS
(SELECT P2.Price, S2.city FROM Parts P2, Supply Y, Suppliers S2
 WHERE P2.pid = Y.sid AND Y.pid = S2.pid
 AND S.sid = S2.sid AND P.pid = P2.pid
 AND P.Price = P2.Price AND S2.city =S.city
)
GROUP BY S.sname
```

Translation of SQL into the relational algebra

Solution

Translation of inner query

```
SELECT P2.Price, S2.city FROM Parts P2, Supply Y, Suppliers S2
WHERE P2.pid = Y.sid AND Y.pid = S2.pid
AND S.sid = S2.sid AND P.pid = P2.pid
AND P.Price = P2.Price AND S2.city =S.city
```

$$e_1 := \boldsymbol{\pi}_{P.*,S.*,P_2.\text{Price},S_2.\text{city}}$$
$$\boldsymbol{\sigma}_{\text{P2.pid} = \text{Y.sid AND Y.pid} = \text{S2.pid AND S.sid} = \text{S2.sid AND P.pid} = \text{P2.pid}}$$
$$\boldsymbol{\sigma}_{\text{P.Price} = \text{P2.Price AND S2.city} = \text{S.city}}$$
$$(\boldsymbol{\rho}_{P_2}\text{Parts} \times \boldsymbol{\rho}_{Y}\text{Supply} \times \boldsymbol{\rho}_{S_2}\text{Suppliers} \times \boldsymbol{\rho}_{P}\text{Parts} \times \boldsymbol{\rho}_{S}\text{Suppliers})$$

Translation of SQL into the relational algebra

Solution

Decorrelation:

```
SELECT MAX(P.price), S.sname
FROM Parts P, Suppliers S
WHERE S.city = Ham
AND EXISTS
(...
)
GROUP BY S.sname
```

$$e_2 := \boldsymbol{\rho}_P \text{Parts} \times \boldsymbol{\rho}_S \text{Suppliers}$$
$$e_3 := \hat{e_2} \bowtie \boldsymbol{\pi}_{P.*,S.*,}(e_1)$$

Note that $\hat{e_2}$ is empty!

Translation of SQL into the relational algebra

Solution

Where, Group, By, and Aggregate:

```
SELECT MAX(P.price), S.sname
FROM Parts P, Suppliers S
WHERE S.city = Ham
AND EXISTS
(...
)
GROUP BY S.sname
```

$$e_4 := \boldsymbol{\sigma}_{S.\text{city}=\text{'Ham'}}(e_3)$$
$$e_5 := \gamma_{S.\text{sname},\text{MAX(S.Price)}}(e_4)$$
$$e_6 := \boldsymbol{\pi}_{S.\text{sname},\text{MAX(S.Price)}}(e_5)$$

Optimization of logical query plans
Eliminating redundant joins

Optimization of select-project-join expressions

Containment of conjunctive queries is decidable

$$A(x, y) \leftarrow R(x, w),\ G(w, z),\ R(z, y)$$
$$B(x, y) \leftarrow R(x, w),\ G(w, w),\ R(w, y)$$

Golden method to check whether $B \subseteq A$:

1. First calculate the canonical database D for B:

$$R: \begin{array}{|cc|}\hline \dot{x} & \dot{w} \\ \dot{w} & \dot{y} \\ \hline \end{array} \qquad G: \begin{array}{|cc|}\hline \dot{w} & \dot{w} \\ \hline \end{array}$$

2. Then check whether $(\dot{x}, \dot{y}) \in A(D)$. If so, $B \subseteq A$, otherwise $B \not\subseteq A$.

Optimization of select-project-join expressions

Containment of conjunctive queries is decidable

$$A(x, y) \leftarrow R(x, w),\ G(w, z),\ R(z, y)$$
$$B(x, y) \leftarrow R(x, w),\ G(w, w),\ R(w, y)$$

Fact: $B \subseteq A \Leftrightarrow (x, y) \in A(D)$ with D the canonical database for B.

First possibility: $(\dot{x}, \dot{y}) \notin A(D)$

In this case we have just constructed a counter-example because $(\dot{x}, \dot{y}) \in B(D)$.

R	
$\dot{x}$	$\dot{w}$
$\dot{w}$	$\dot{y}$

G	
$\dot{w}$	$\dot{w}$

Optimization of select-project-join expressions

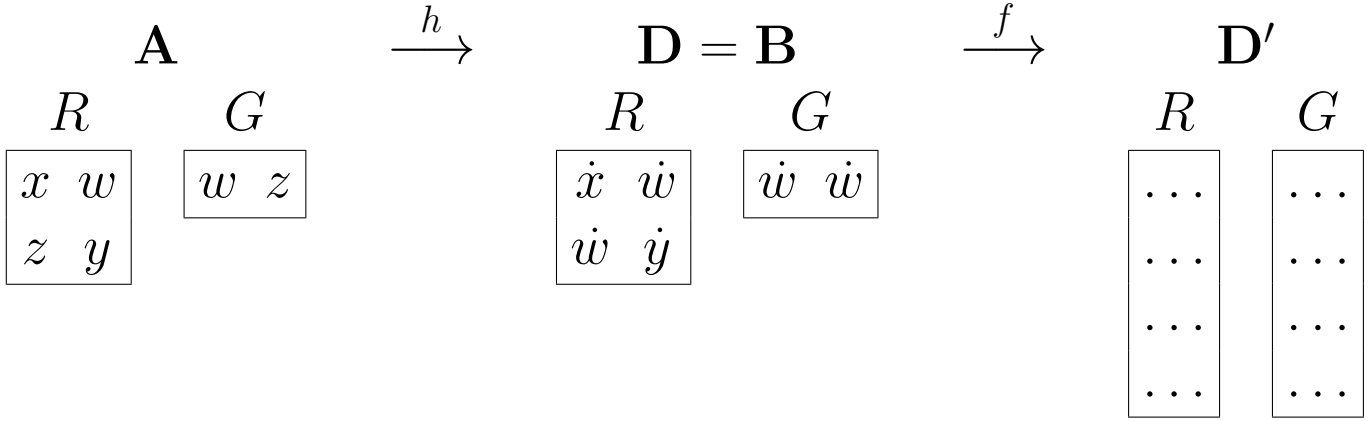
Containment of conjunctive queries is decidable

$$A(x, y) \leftarrow R(x, w),\ G(w, z),\ R(z, y)$$
$$B(x, y) \leftarrow R(x, w),\ G(w, w),\ R(w, y)$$

Fact: $B \subseteq A \Leftrightarrow (x, y) \in A(D)$ with D the canonical database for B.

Second possibility: $(\dot{x}, \dot{y}) \in A(D)$

- There hence exists a matching h of A into D such that $h(x) = \dot{x}$ and $h(y) = \dot{y}$
- Let D' be an arbitrary other database, and pick $t \in B(D')$. There hence exists a matching f such that $t = (f(x), f(y))$.
- Then $f \circ h$ is a matching of A on D':



Optimization of select-project-join expressions

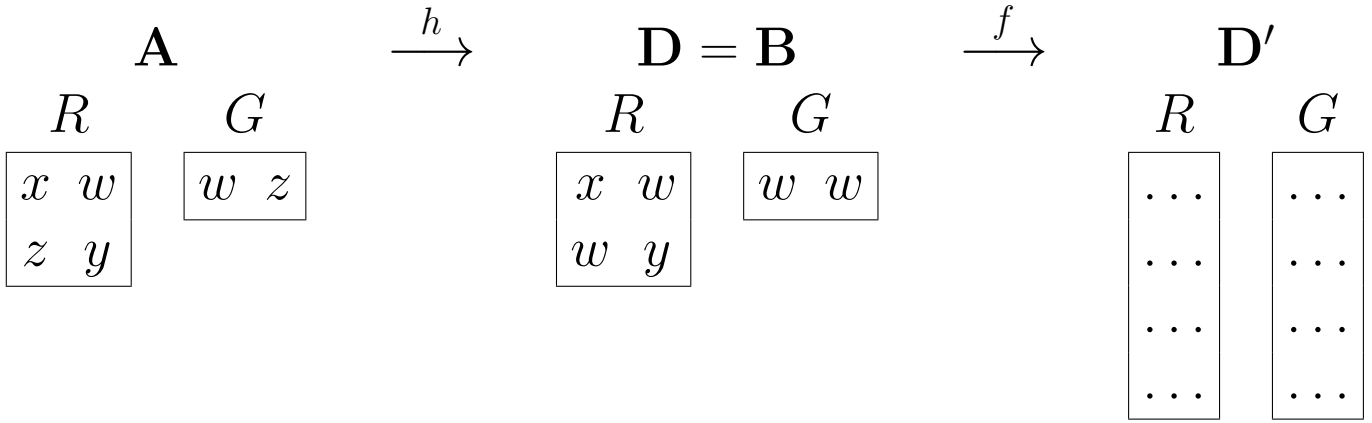
Containment of conjunctive queries is decidable

$$A(x, y) \leftarrow R(x, w),\ G(w, z),\ R(z, y)$$
$$B(x, y) \leftarrow R(x, w),\ G(w, w),\ R(w, y)$$

Fact: $B \subseteq A \Leftrightarrow (x, y) \in A(D)$ with D the canonical database for B.

Second possibility: $(\dot{x}, \dot{y}) \in A(D)$

- There hence exists a matching h of A into D such that $h(x) = \dot{x}$ and $h(y) = \dot{y}$
- Let D' be an arbitrary other database, and pick $t \in B(D')$. There hence exists a matching f such that $t = (f(x), f(y))$.
- Then $f \circ h$ is a matching of A on D':



Optimization of select-project-join expressions

Containment of conjunctive queries is decidable

$$A(x, y) \leftarrow R(x, w),\ G(w, z),\ R(z, y)$$
$$B(x, y) \leftarrow R(x, w),\ G(w, w),\ R(w, y)$$

Fact: $B \subseteq A \Leftrightarrow (x, y) \in A(D)$ with D the canonical database for B.

Second possibility: $(x, y) \in A(D)$

- There hence exists a matching h of A into D such that $h(x) = x$ and $h(y) = y$
- Let D' be an arbitrary other database, and pick $t \in B(D')$. There hence exists a matching f such that $t = (f(x), f(y))$.
- Then $f \circ h$ is a matching of A on D':
- And hence

$$t = (f(x), f(y)) = (f(h(x)), f(h(y)) \in A(D')$$

Optimization of select-project-join expressions

Conclusion:

- Containment of conjunctive queries is decidable
- Consequently the equivalence of conjunctive queries is also decidable

Optimization of select-project-join expressions

Optimizing conjunctive queries

Input: A conjunctive query Q

Output: A conjunctive query Q' equivalent to Q that is optimal (i.e., has the least number of atoms in its body).

For each conjunctive query we can obtain an equivalent, optimal query, by removing atoms from its body

- Let Q be a CQ and let P be an arbitrary optimal and equivalent query.
- Then $Q \subseteq P$ and hence $(\dot{x}, \dot{y}) \in P(D_Q)$ with D_Q the canonical database for Q. Let f be the matching that ensures this fact.
- Let Q' be obtained by removing from Q all atoms that are not in the range of f
- Then $Q \subseteq Q'$
- Moreover, also $Q' \subseteq P$ (because $(\dot{x}, \dot{y}) \in P(D_{Q'})$ still holds) and $P \subseteq Q$. Hence $Q' \equiv Q$.
- Note that Q' contains at most the same number of atoms as P. Hence Q' is optimal.

Optimization of select-project-join expressions

Optimization of conjunctive queries

Input: A conjunctive query Q

Output: A conjunctive query Q' equivalent to Q that is optimal (i.e., has the least number of atoms in its body).

Optimization algorithm

- A conjunctive query is given. Consider for example:

  $$Q(x) \leftarrow R(x, x),\ R(x, y)$$

- We check, atom by atom, what atoms in its body are redundant.

  We next try to remove $R(x, y)$:

  $$Q_2(x) \leftarrow R(x, x)$$

  Note that $Q \subseteq Q_2$ and $Q_2 \subseteq Q$.

  Q_2 is certainly shorter than Q and hence closer to the optimal query. Since there remain no other atoms to test, our result is Q_2.

Optimization of select-project-join expressions

Optimization of select-project-join expressions

1. Translate the select-project-join expression e into an conjunctive query Q.
2. Optimize Q.
3. Translate Q back into a select-project-join expression.

Optimization of arbitrary relational algebra

Undecidable in general!

Our method in integrated exercises:

1. Identify syntactically maximal subexpressions that are SPJ
2. Optimize these

Example

$$\boldsymbol{\pi}_{D.\texttt{floor}}\boldsymbol{\sigma}_{E.did=D.did}($$
$$[\boldsymbol{\pi}_{D.*,E.*}\boldsymbol{\sigma}_{F_2.\texttt{did}=E.\texttt{did}\wedge E_2.\texttt{did}=D.\texttt{did}\wedge E_2.\texttt{eid}=E.\texttt{eid}\wedge F_2.\texttt{expenses}=300\wedge E.\texttt{did}=F_2.\texttt{did}}$$
$$(\boldsymbol{\rho}_D(\texttt{Dept}) \times \boldsymbol{\rho}_E(\texttt{Emp}) \times \boldsymbol{\rho}_{F2}(\texttt{Finance}) \times \boldsymbol{\rho}_{E2}(\texttt{Emp}))]$$
$$\bowtie [\boldsymbol{\pi}_{D.*}\boldsymbol{\sigma}_{F_1.\texttt{budget}>150\wedge D_2.\texttt{did}=F_1.\texttt{did}\wedge D_2.\texttt{floor}=D.\texttt{floor}}$$
$$(\boldsymbol{\rho}_D(\texttt{Dept}) \times \boldsymbol{\rho}_{D2}(\texttt{Dept}) \times \boldsymbol{\rho}_{F1}(\texttt{Finance}))]$$

One-dimensional index structures

Linear Hashing

Q: When do we increase ?

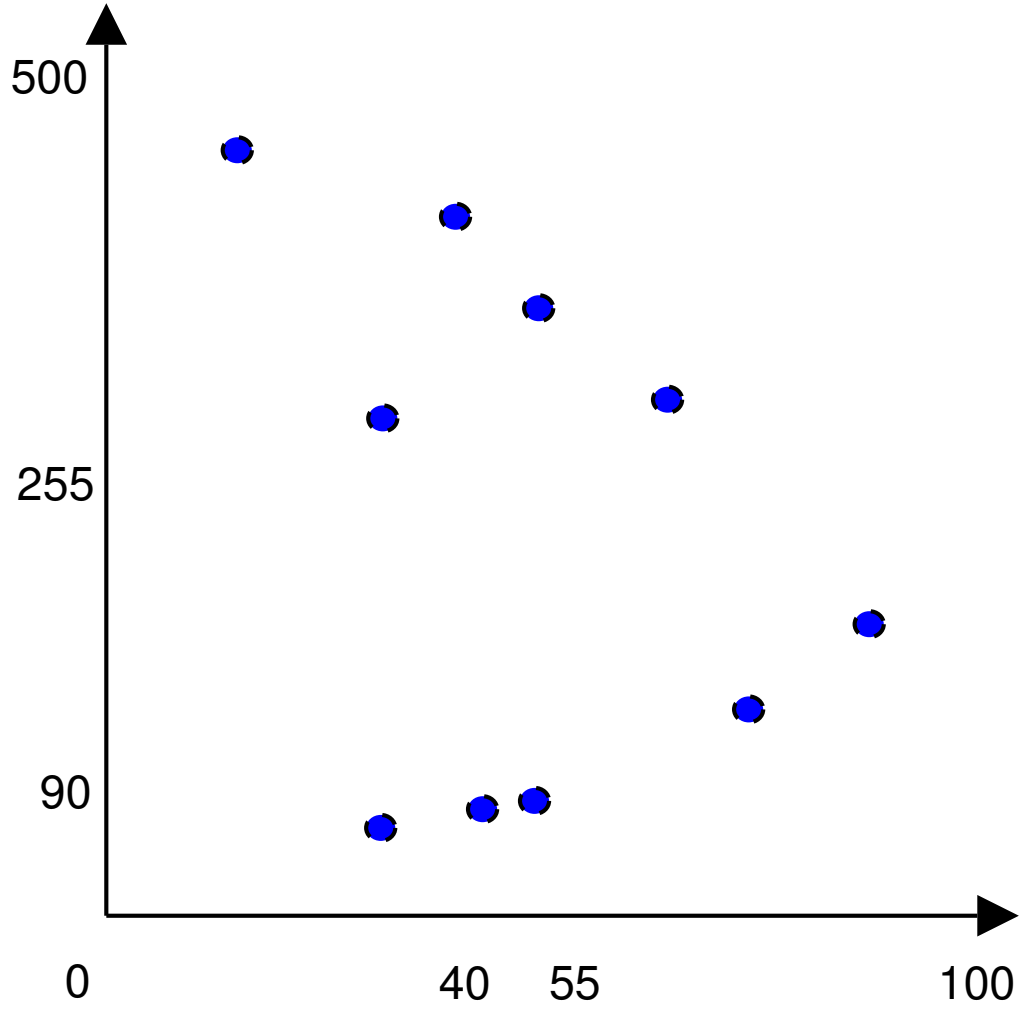
A: See corresponding slides

Multi-dimensional index structures

Part I: motivation

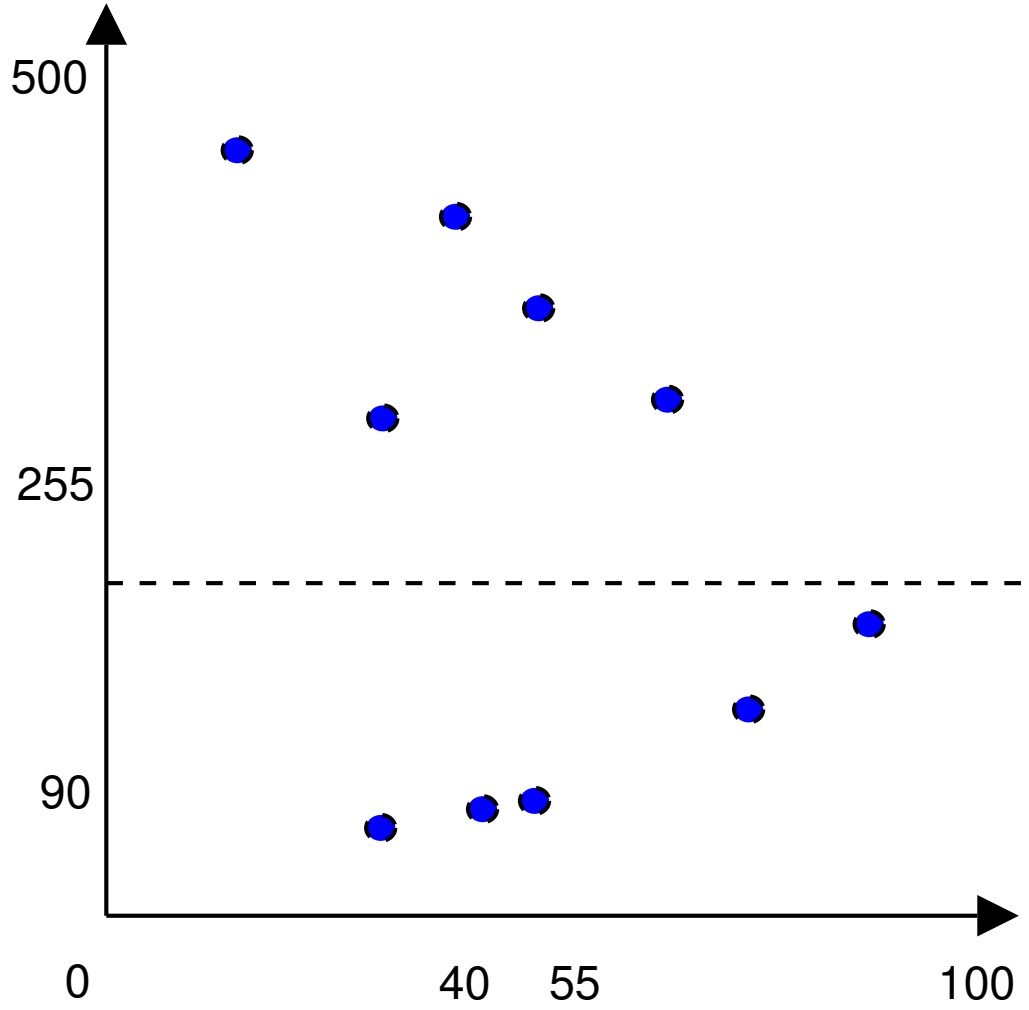
Multidimensional Indexes

kd-Trees



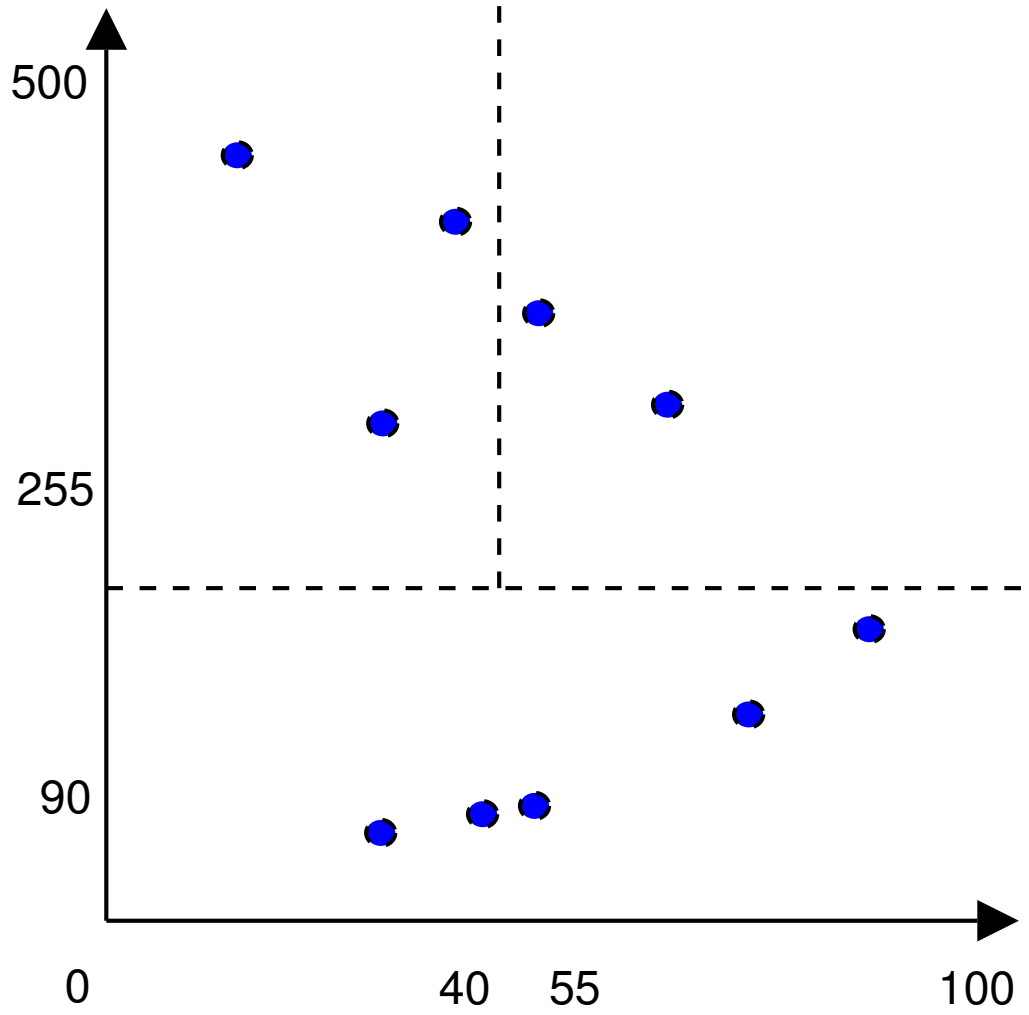
Multidimensional Indexes

kd-Trees



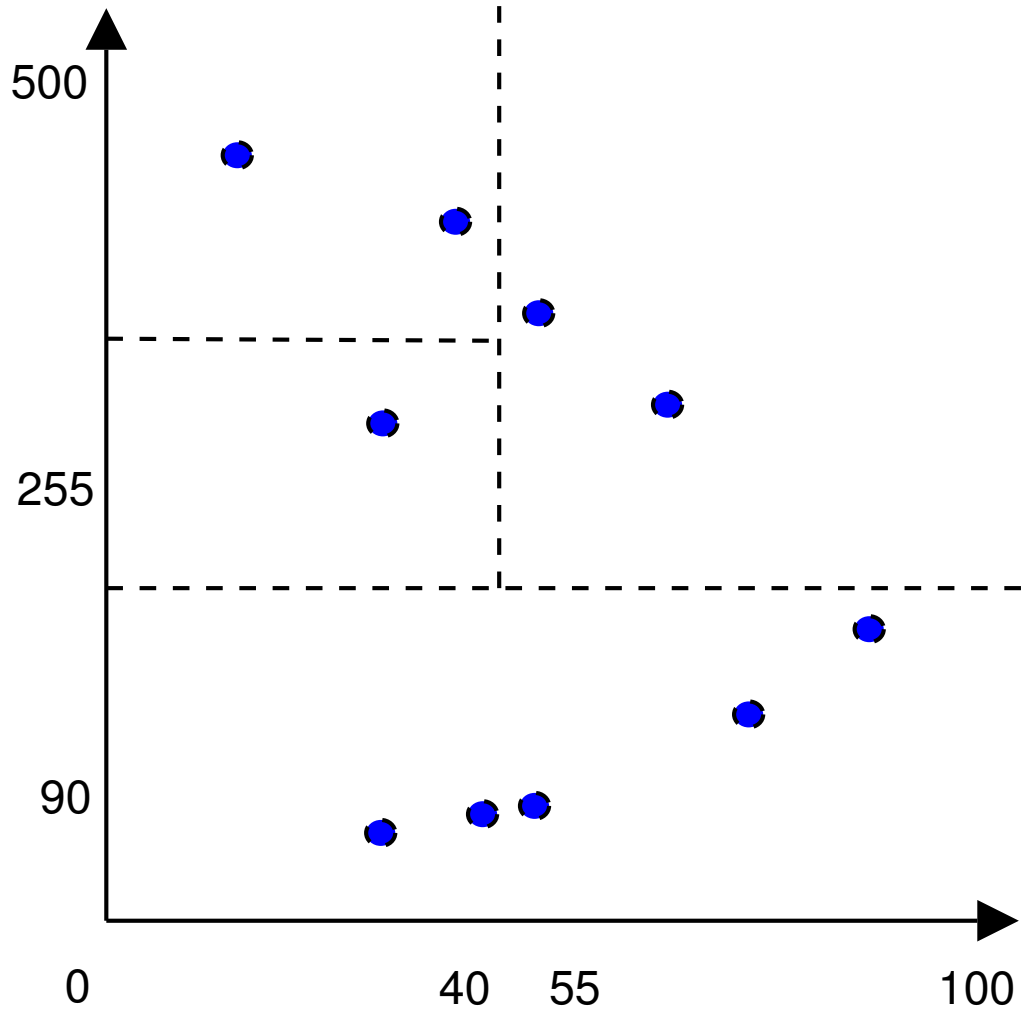
Multidimensional Indexes

kd-Trees



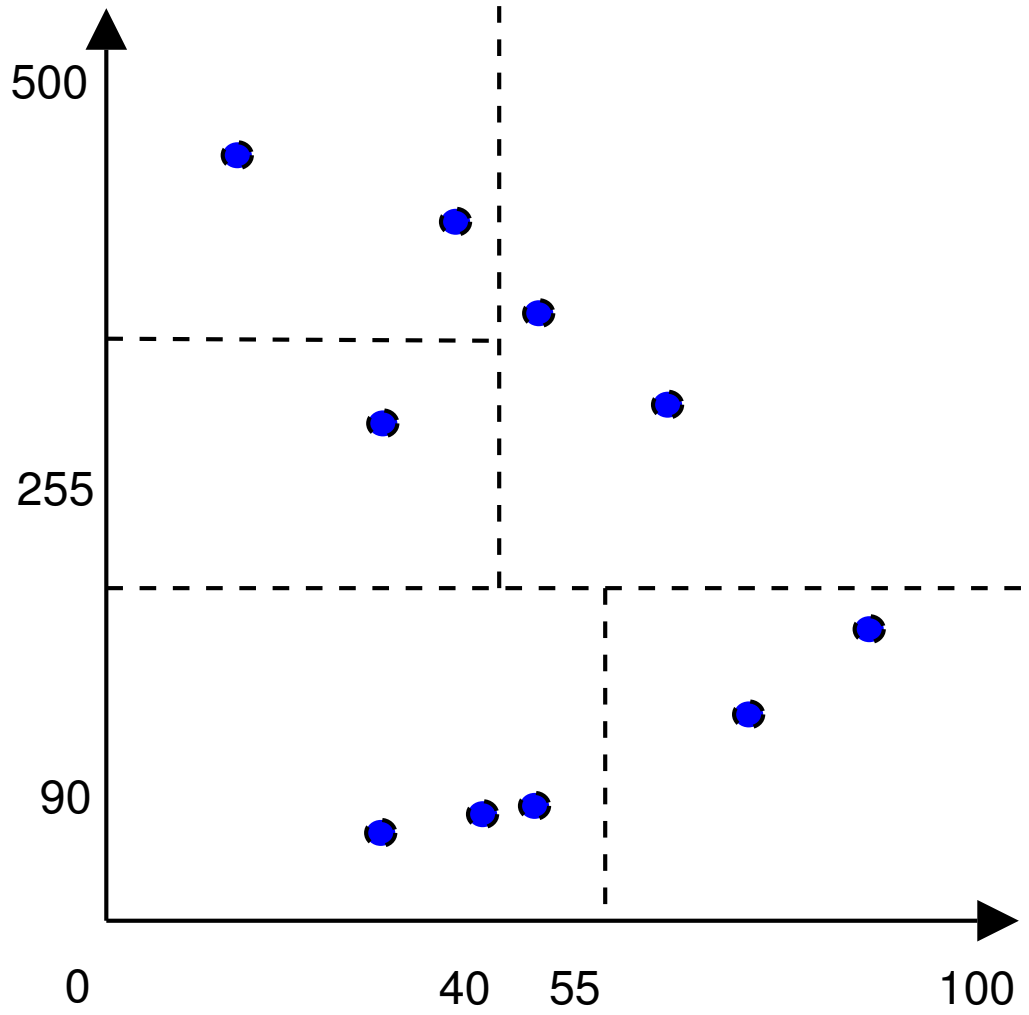
Multidimensional Indexes

kd-Trees



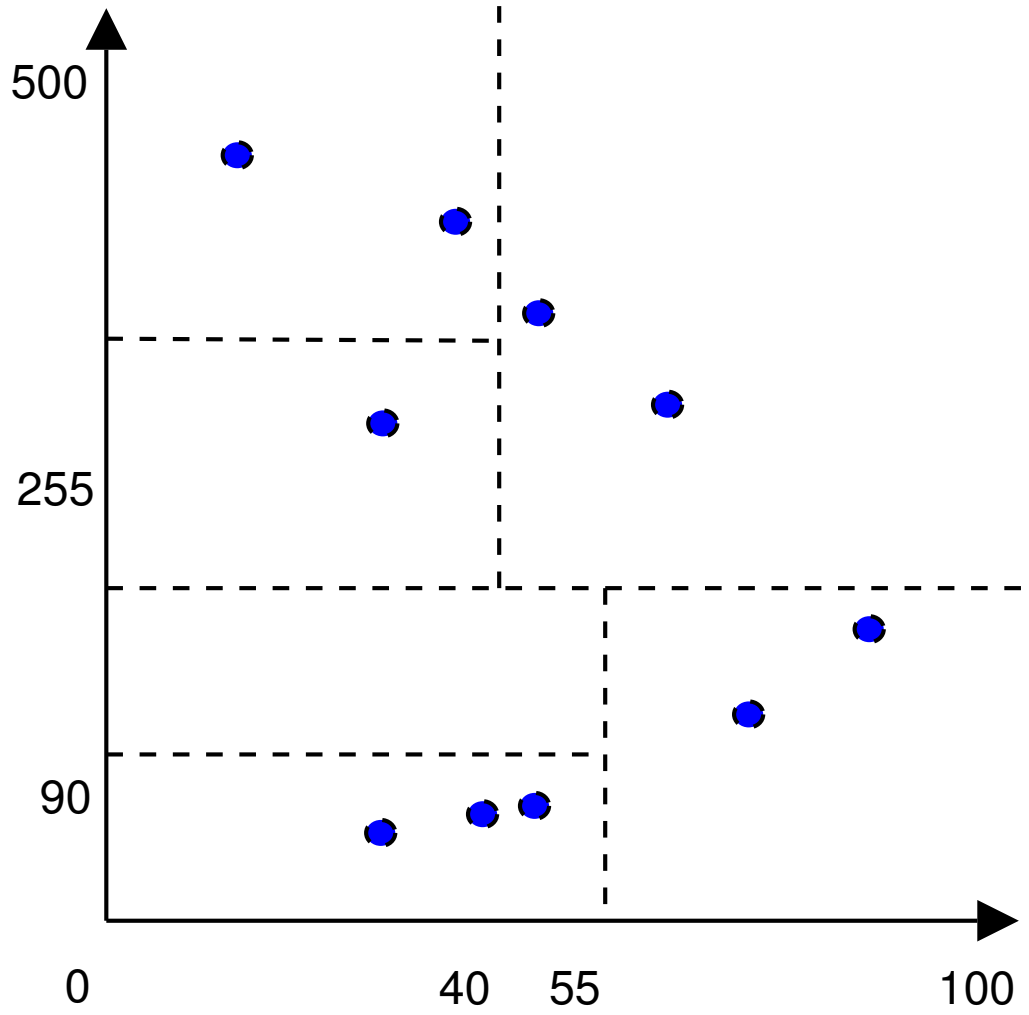
Multidimensional Indexes

kd-Trees



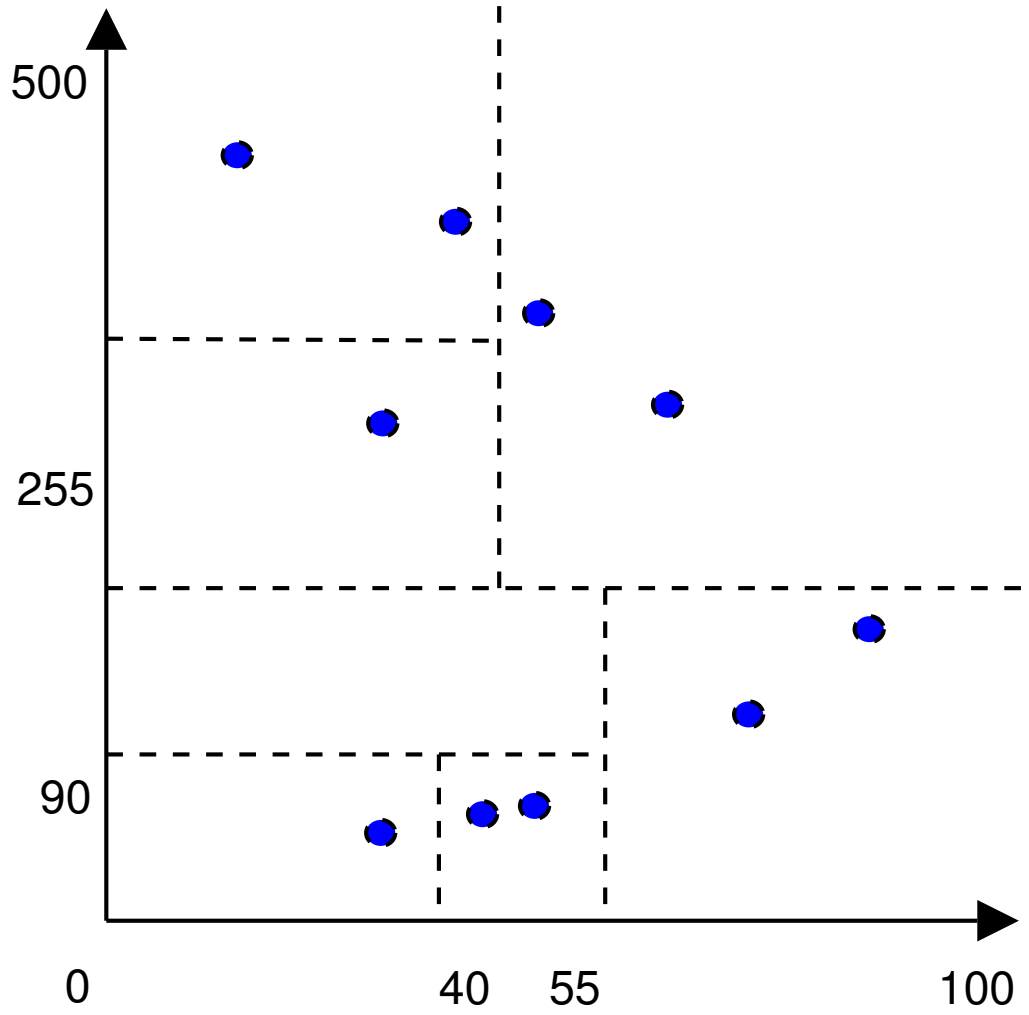
Multidimensional Indexes

kd-Trees



Multidimensional Indexes

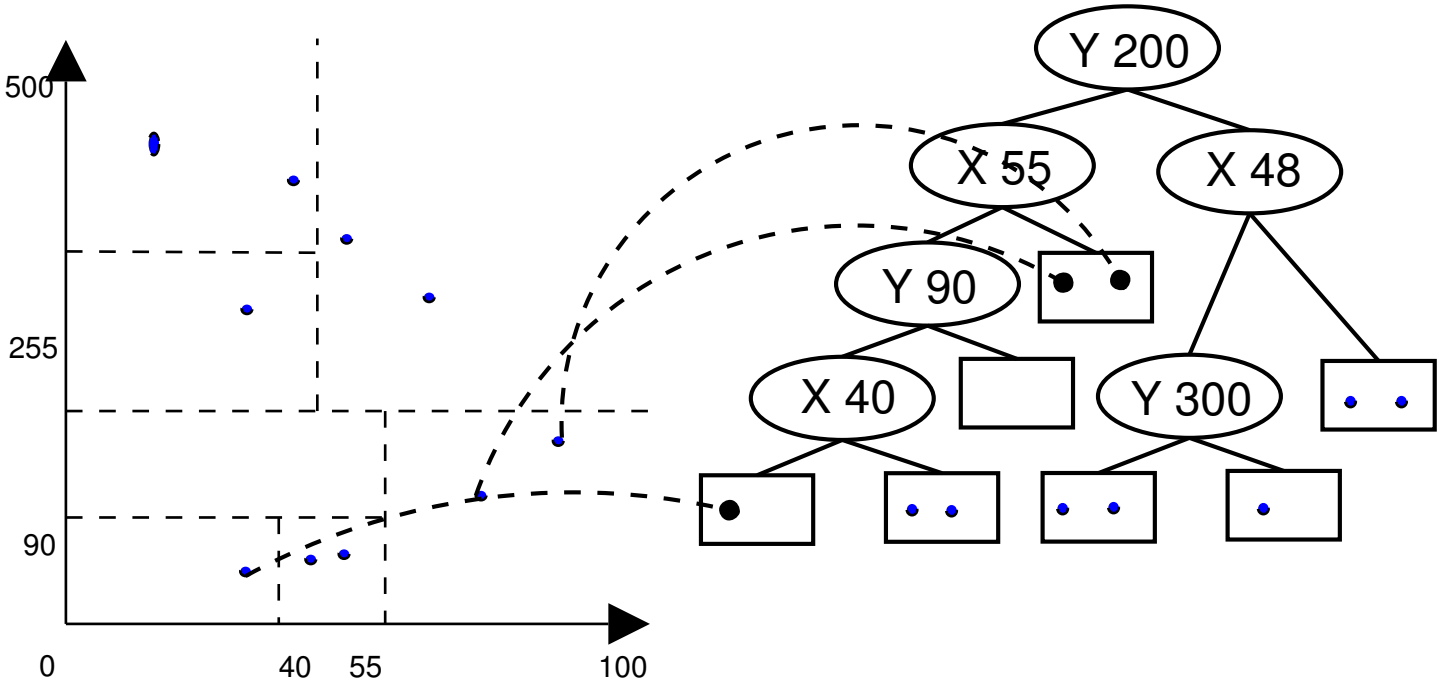
kd-Trees



Multidimensional Indexes

kd-Trees

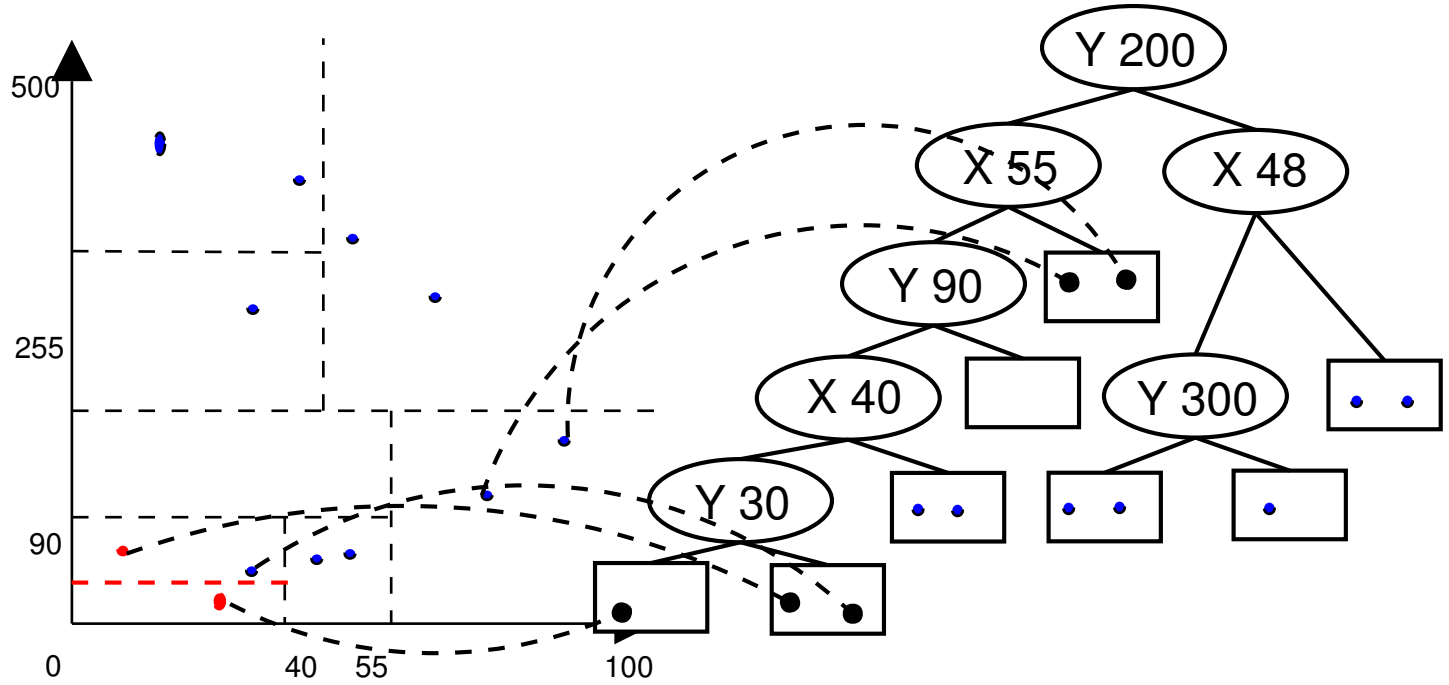
We can look at this as a tree as follows:



Multidimensional Indexes

kd-Trees

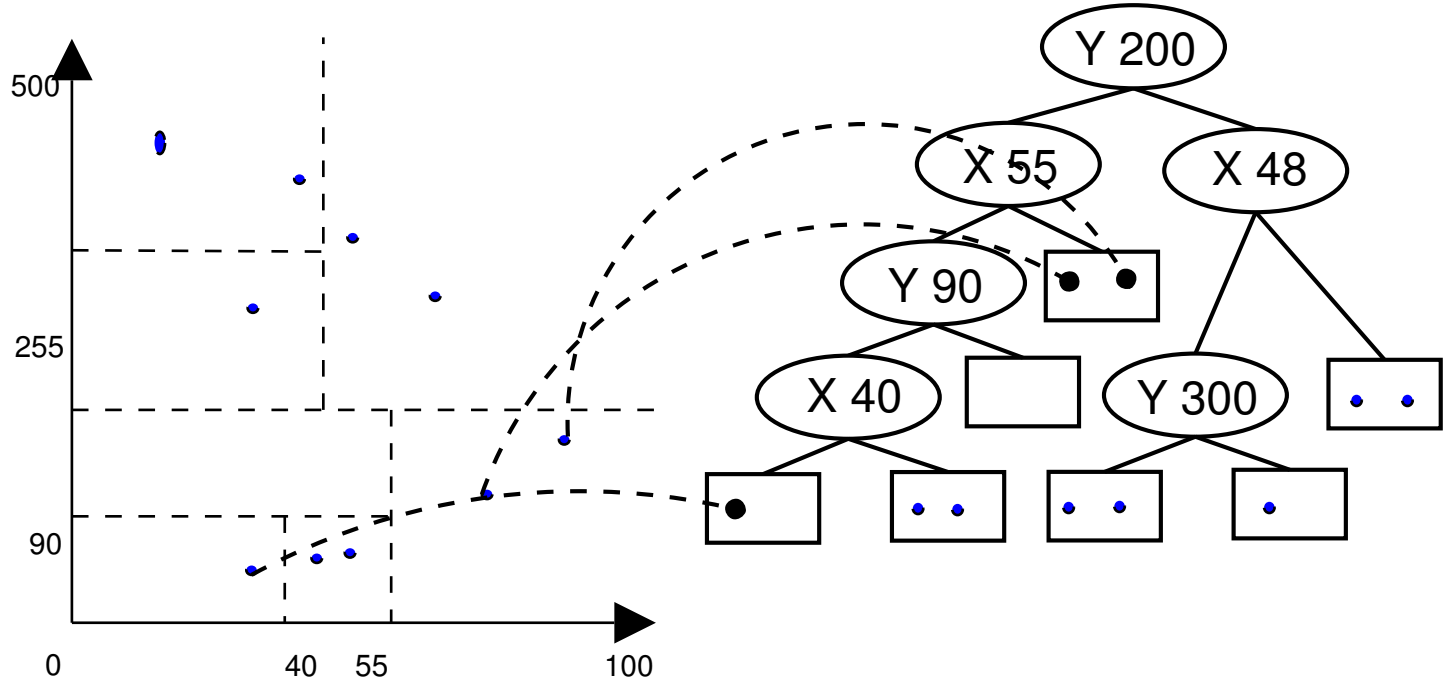
We continue splitting after new insertions:



Multidimensional Indexes

kd-Trees

- Good support for point queries
- Good support for partial match queries: e.g., $(y = 40)$
- Good support for range queries $(40 \leq x \leq 45 \wedge y < 80)$
- Reasonable support for nearest neighbour



Multidimensional Indexes

kd-Trees for secondary storage

- Generalization to n children for each interal node (cf. BTree).

  But it is difficult to keep this tree balanced since we cannot merge the children

- We limit ourselves to two children per node (as before), but store multiple nodes in a single block.

Multidimensional Indexes

R-Trees: generalization of BTrees

Designed to index regions (where a single point is also viewed as a region).

See illustration at `https://www.youtube.com/watch?v=u6SUeQtKBsY`

Physical Operators

Scanning, sorting, merging, hashing

Physical Operators

One-pass set union

Assume that $M - 1 \geq B(R)$. We can then compute the set union $R \cup_S S$ as follows (R and S are assumed to be sets themselves)

load R into memory buffers $N_1, \ldots, N_{B(R)}$;
  for each tuple t_R in $N_1, \ldots, N_{B(R)}$ **do**
    output t_R
for each block B_S in S **do**
  load B_S into buffer N_0;
  for each tuple t_S in N_0 **do**
    if t_S does not occur in $N_1, \ldots, N_{B(R)}$
      output t_S

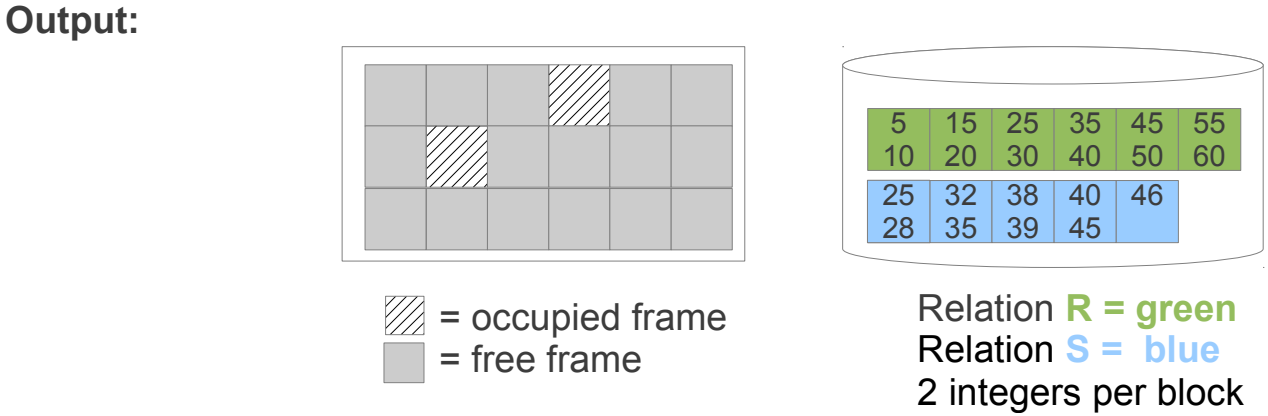
- Cost: $B(R) + B(S)$ I/O operations (ignoring output-cost)
- Note that it also costs time to check whether t_S occurs in $N_1, \ldots, N_{B(R)}$. By using a suitable main-memory data structure this can be done in $O(n)$ or $O(n \log n)$ time. We ignore this cost.
- Requires $B(R) \leq M - 1$

Physical Operators

Sort-based set union

We can also alternatively compute the set union $R \cup_S S$ as follows (again R and S are assumed to be sets):

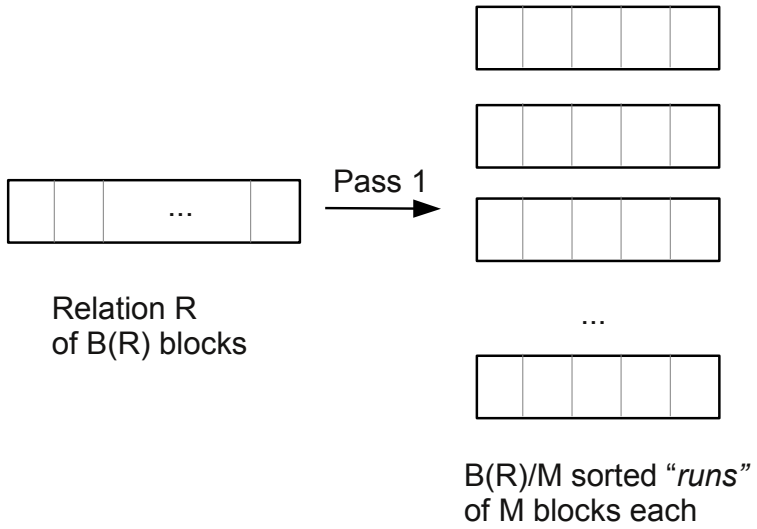
1. Sort R
2. Sort S
3. Iterate synchronously over R and S, at each point loading 1 block of each relation in memory and inspecting 1 tuple of R and S.



Physical Operators

Sort-based set union

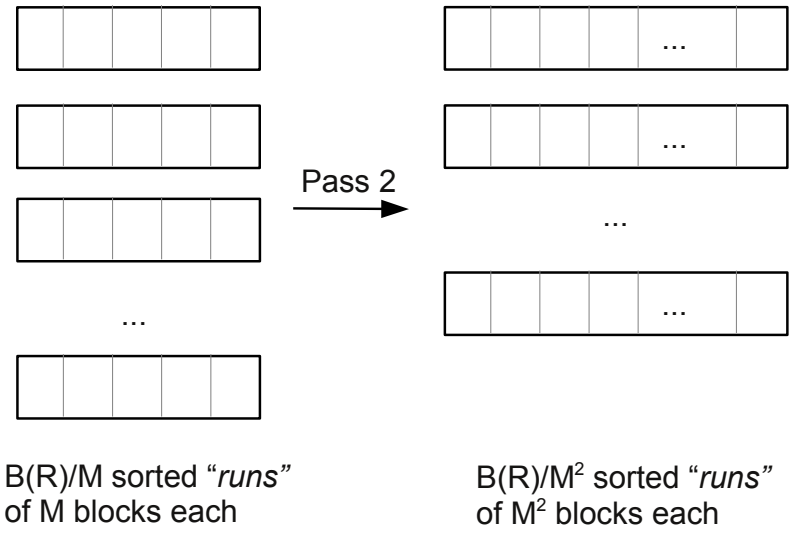
- Sorting can in principle be done by any suitable algorithm, but is usually done by Multiway Merge-Sort:
  - In the first pass we read M blocks at the same time from the input relation, sort these by means of a main-memory sorting algorithm, and write the sorted resulting sublist to disk. After the first pass we hence have $B(R)/M$ sorted sublists of M blocks each.



Physical Operators

Sort-based set union

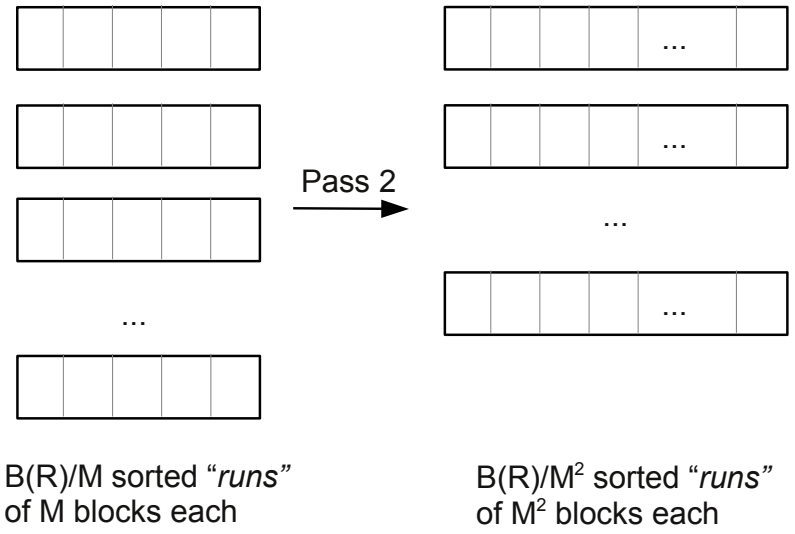
- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  - In the 2nd pass, we merge the first M sublists from the first pass into a single sublist of M^2 blocks. We do so by iterating synchronously over these M sublists, keeping 1 block of each list into memory during this iteration.



Physical Operators

Sort-based set union

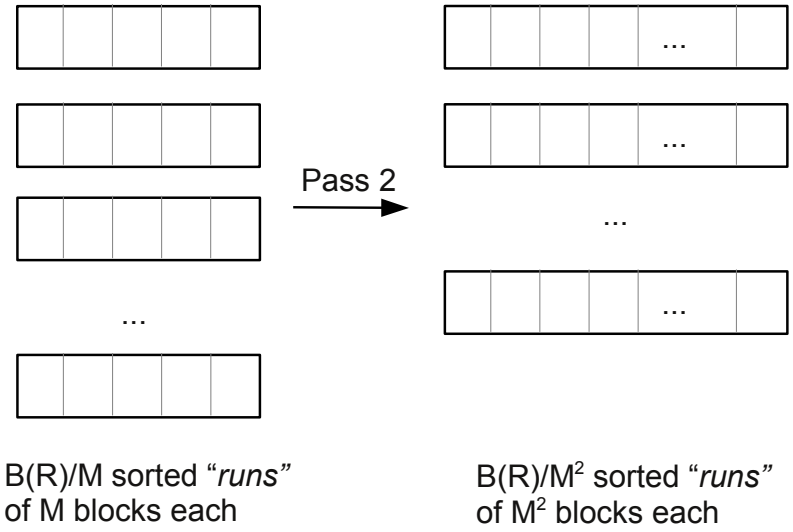
- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  - We then merge the next M sublists into a single sublist, and continue until we have treated each sublist resulting from the first pass.



Physical Operators

Sort-based set union

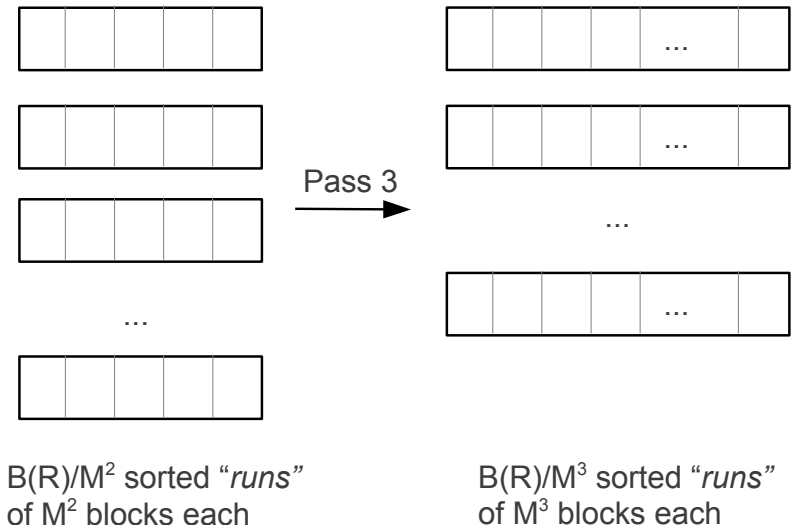
- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  - After the second pass we hence have $B(R)/M^2$ sorted sublists of M^2 blocks each.



Physical Operators

Sort-based set union

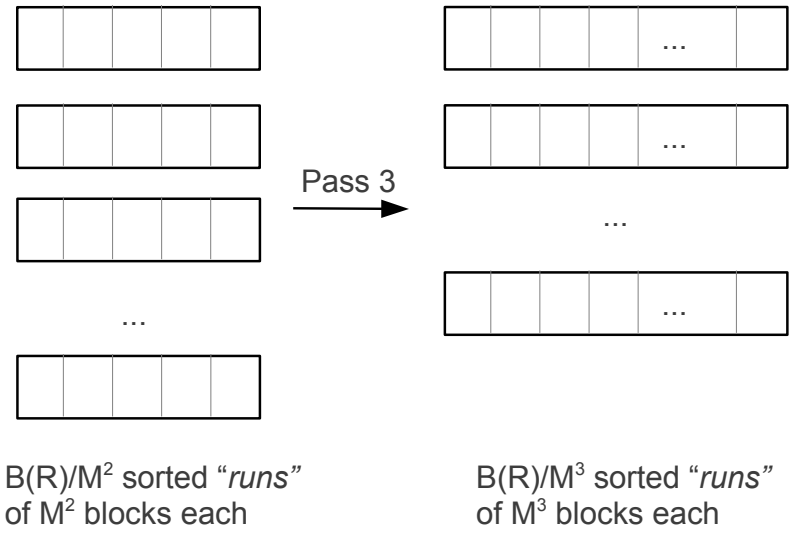
- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  - In the 3rd pass, we merge the first M sublists from the 2nd pass (each of M^2 blocks) into a single sublist of M^3 blocks. We do so by iterating synchronously over these M sublists, keeping 1 block of each list into memory during this iteration.



Physical Operators

Sort-based set union

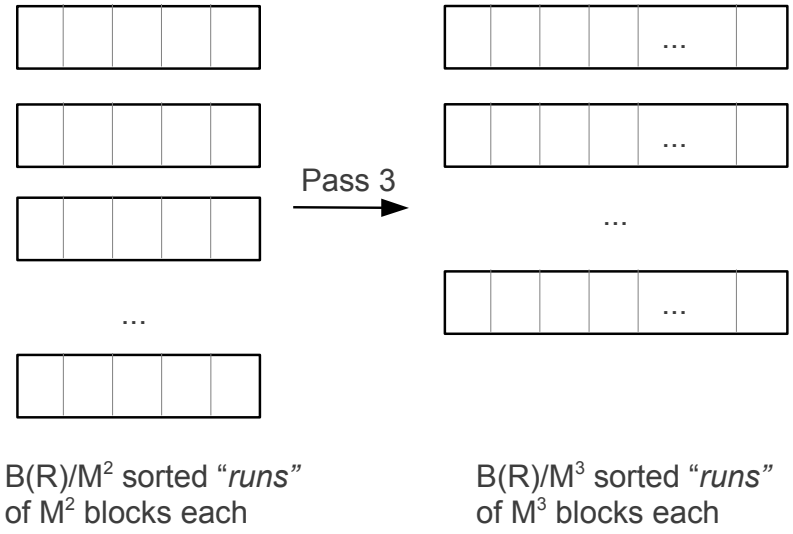
- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  - We then merge the next M sublists into a single sublist, and continue until we have treated each sublist resulting from the 2nd pass .



Physical Operators

Sort-based set union

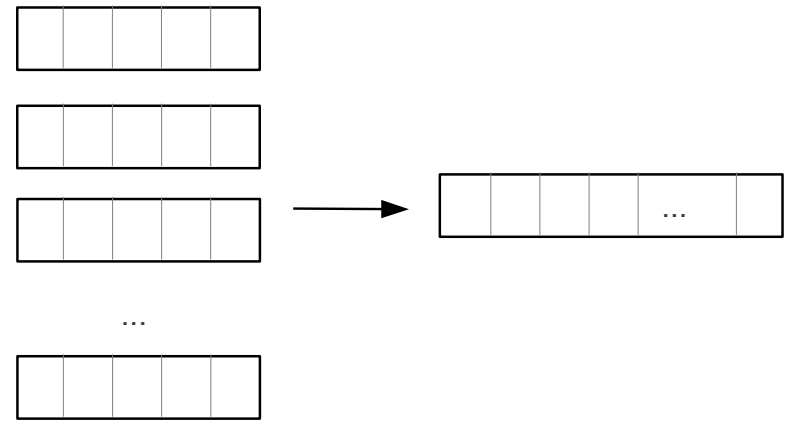
- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  - After the 3rd pass we hence have $B(R)/M^3$ sorted sublists of M^3 blocks each.



Physical Operators

Sort-based set union

- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  - We keep doing new passes until we reach a single sorted list.



Physical Operators

Sort-based set union

- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  1. In the first pass we read M blocks at the same time from the input relation, sort these by means of a main-memory sorting algorithm, and write the sorted resulting sublist to disk. After the first pass we hence have $B(R)/M$ sorted sublists of M blocks each.
  2. In the following passes we keep reading M blocks from these sublists and merge them into larger sorted sublists. (After the second pass we hence have $B(R)/M^2$ sorted sublists of M^2 blocks each, after the third pass $B(R)/M^3$ sorted sublists, ...)
  3. We repeat until we obtain a single sorted sublist.
- What is the complexity of this?
  1. In each pass we read and write the entire input relation exactly once.
  2. There are $\lceil \log_M B(R) \rceil$ passes
  3. The total cost is hence $2B(R) \lceil \log_M B(R) \rceil$ I/O operations.

Physical Operators

Sort-based set union

- The costs of sort-based set union:
  1. Sorting R : $2B(R)\lceil \log_M B(R) \rceil$ I/O's
  2. Sorting S : $2B(S)\lceil \log_M B(S) \rceil$ I/O's
  3. Synchronized iteration: $B(R) + B(S)$ I/O's

  In Total:

$$\boxed{2B(R)\lceil \log_M B(R) \rceil + 2B(S)\lceil \log_M B(S) \rceil + B(R) + B(S)}$$

- Uses M memory-buffers during sorting
- Requires 2 memory-buffers for synchronized iteration

Physical Operators

Sort-based set union

Remark: the "synchronized iteration" phase of sort-based set union is very similar to the merge phase of multiway merge-sort. Sometimes it is possible to combine the last merge phase with the synchronized iteration, and avoid $2B(R) + 2B(S)$ I/Os:

1. Sort R, but do not execute the last merge phase. R is hence still divided in $1 < l \leq M$ sorted sublists.
2. Sort S, but do not execute the last merge phase. S is hence still divided in $1 < k \leq M$ sorted sublists.
3. If $l + k < M$ then we can use the M available buffers to load the first block of each sublist of R and S in memory.
4. Then iterate synchronously through these sublists: at each point search the "smallest" (according to the sort order) record in the $l + k$ buffers, and output that. Move to the next record in the buffers when required. When all records from a certain buffer are processed, load the next block from the corresponding sublist.

Physical Operators

Sort-based set union

The cost of the optimized sort-based set union algorithm is as follows:

1. Sort R, but do not execute the last merge phase.

$$2B(R)(\lceil \log_M B(R) \rceil - 1)$$

2. Sort S, but do not execute the last merge phase.

$$2B(S)(\lceil \log_M B(S) \rceil - 1)$$

3. Synchronized iteration through the sublists: $B(R) + B(S)$ I/O's

Total:

$$\boxed{2B(R)\lceil \log_M B(R) \rceil + 2B(S)\lceil \log_M B(S) \rceil - B(R) - B(S)}$$

We hence save $2B(R) + 2B(S)$ I/O's.

Physical Operators

Sort-based set union

Note that this optimization is only possible if $k + l \leq M$.

Observe that $k = \left\lceil \frac{B(R)}{M^{\lceil \log_M B(R) \rceil - 1}} \right\rceil$ and $l = \left\lceil \frac{B(S)}{M^{\lceil \log_M B(S) \rceil - 1}} \right\rceil$.

In other words, this optimization is only possible if:

$$\left\lceil \frac{B(R)}{M^{\lceil \log_M B(R) \rceil - 1}} \right\rceil + \left\lceil \frac{B(S)}{M^{\lceil \log_M B(S) \rceil - 1}} \right\rceil \leq M$$

Physical Operators

Hash-based set union

We can also alternatively compute the set union $R \cup_S S$ as follows (R and S are assumed to be sets, and we assume that $B(R) \leq B(S)$):

1. Partition, by means of hash function(s), R in buckets of at most $M-1$ blocks each. Let k be the resulting number of buckets, and let R_i be the relation formed by the records in bucket i.
2. Partition, by means of the same hash function(s) as above, S in k buckets. Let S_i be the relation formed by the records in bucket i.

   Observe: the records in R_i and S_i have the same hash value! A record t hence occurs in both R and S if, and only if, there is a bucket i such that t occurs in both R_i and S_i.
3. We can hence compute the set union by calculating the set union of R_i and S_i, for every $i \in 1, \ldots, k$. Since every R_i contains at most $M-1$ blocks, we can do so using the one-pass algorithm.

Note: in contrast to the sort-based set union, the output of a hash-based set union is unsorted!

Physical Operators

Hash-based set union

How do we partition R in buckets of at most $M-1$ blocks?

1. Using M buffers, we first hash R into $M-1$ buckets.
2. Subsequently we partition each bucket separately in $M-1$ new buckets, by using a new hash function distinct from the one used in the previous step (why?)
3. We continue doing so until the obtained buckets consists of at most $M-1$ blocks.

Physical Operators

Hash-based set union

What is the cost of partitioning?

1. Assuming that the hash function(s) distribute the records uniformly, we have $M-1$ buckets of $\frac{B(R)}{M-1}$ blocks after the first pass, $(M-1)^2$ buckets of $\frac{B(R)}{(M-1)^2}$ blocks after the second pass, and so on. Hence, if we reach buckets of at most $M-1$ blocks after k passes, k must satisfy:

$$\frac{B(R)}{(M-1)^k} \leq M-1$$

   The minimal value of k that satisfies this is hence $\lceil \log_{M-1} B(R) - 1 \rceil$

2. In every pass we read and write R once.

Total cost:

$$\boxed{2B(R)\lceil \log_{M-1} B(R) - 1 \rceil}$$

Physical Operators

Hash-based set union

What is the costs of calculating hash-based set union?

1. Partition R: $2B(R)\left\lceil \log_{M-1} B(R) - 1 \right\rceil$ I/O's
2. Partition S: $2B(S)\left\lceil \log_{M-1} B(R) - 1 \right\rceil$ I/O's

   Because we "only" need to partition S in as many buckets as R.
3. The one-pass set union of each R_i and S_i: $B(R) + B(S)$

Total:

$$\boxed{2B(R)\left\lceil \log_{M-1} B(R) - 1 \right\rceil + 2B(S)\left\lceil \log_{M-1} B(R) - 1 \right\rceil + B(R) + B(S)}$$

Physical Operators

Sort-merge Join

Essentially the same algorithm as sort-based set union:

1. Sort R on attribute Y
2. Sort S on attribute Y
3. Iterated synchronously through R and S, keeping 1 block of each relation in memory at all times, and at each point inspecting a single tuple from R and S. Assume that we are currently at tuple t_R in R and at tuple t_S in S.
   - If $t_R.Y < t_S.Y$ then we advance the pointer t_R to the next tuple in R (possibly loading the next block of R if necessary).
   - If $t_R.Y > t_S.Y$ then we advance the pointer t_S to the next tuple in S (possibly loading the next block of S if necessary)).
   - If $t_R.Y = t_S.Y$ then we output $t_R \bowtie t'_S$ for each tuple t'_S following t_S (including t_S itself) that satisfies $t'_S.Y = t_S.Y$. It is possible that we need to read the following blocks in S. Finally, we advance t_R to the next tuple in R, and rewind our pointer in S to t_S.

Physical Operators

Sort-merge Join

- The cost depends on the number of tuples with equal values for Y. The worst case is when all tuples in R and S have the same Y-value. The cost is then $B(R) \times B(S)$ plus the cost for sorting R and S.
- However, joins are often performed on foreign key attributes. Assume for example that attribute Y in S is a foreign key to attribute Y in R. Then every value for Y in S has only one matching tuple in R, and there is no need to reset the pointer in S.
- In this case the cost analysis is similar to the analysis for sort-based set union.

$$\boxed{2B(R)\lceil \log_M B(R)\rceil + 2B(S)\lceil \log_M B(S)\rceil + B(R) + B(S)}$$

- Similarly, it is possible to optimize and gain $2B(R) + 2B(S)$ I/O operations

$$\boxed{2B(R)\lceil \log_M B(R)\rceil + 2B(S)\lceil \log_M B(S)\rceil - B(R) - B(S)}$$

provided that

$$\left\lceil \frac{B(R)}{M^{\lceil \log_M B(R)\rceil - 1}} \right\rceil + \left\lceil \frac{B(S)}{M^{\lceil \log_M B(S)\rceil - 1}} \right\rceil \leq M$$

Physical Operators

Hash-Join

Essentially the same algorithm as hash-based set union:

1. Partition, by hashing the Y-attribute, R into buckets of at most $M-1$ blocks each. Let k be the number of buckets required, and let R_i be the relation formed by the blocks in bucket i.
2. Partition, by hashing the Y-attribute using the same has function(s) as above, S into k buckets. Let S_i be the relation formed by the blocks in bucket i.

   Notice: the records in R_i and S_i have the same hash value. A tuple $t_R \in R$ hence matches the Y attribute of tuple $t_S \in S$ if, and only if, there is a bucket i such that $t_R \in R_i$ and $t_S \in S_i$.
3. We can therefore compute the join by calculating the join of R_i and S_i, for every $i \in 1, \ldots, k$. Since every R_i consists of at most $M-1$ blocks, this can be done using the one-pass algorithm.

Remark: the output of a hash-join is unsorted on the Y attribute, in contrast to the output of the sort-merge join!

Physical Operators

Hash-Join

- Assuming foreign-key joins, the cost analysis is the same as the analysis for hash-based set union
  1. Partition R: $2B(R)\lceil \log_{M-1} B(R) - 1 \rceil$ I/O's
  2. Partition S: $2B(S)\lceil \log_{M-1} B(R) - 1 \rceil$ I/O's

     Because we "only" need to partition S in as many buckets as R.
  3. The one-pass set union of each R_i and S_i: $B(R) + B(S)$

  Total:

  $$\boxed{2B(R)\lceil \log_{M-1} B(R) - 1 \rceil + 2B(S)\lceil \log_{M-1} B(R) - 1 \rceil + B(R) + B(S)}$$

- Again the book focuses on "two-pass hash-join":

  one pass for the partitioning, one pass for the join

Cost-Based Plan Selection

Enumerate, Estimate, Select

Cost-Based Plan Selection

Greedy plan selection

In the exercises we will use the following greedy algorithm.

- Start with a logical query plan without join ordering.
- We work bottom-up: first we assign physical operators to the leaves, then to the parents of the leaves, then to their parents, and so on. At each point we choose the phyiscal operator with the least cost. At each point, if the operator can be pipelined we decide to pipeline, otherwise we materialize.
- When we reach a join operator (e.g., $R \bowtie S \bowtie T \bowtie U$) and need to determine an ordering of its various members then:
  1. We start by joining the two relations for which the best physical join algorithm yields the smallest cost
     $\rightarrow$ e.g., execute $R \bowtie T$ through a hash-join
  2. Add, from the remaining relations (S or U), those relations to the join for which the best physical join-algorithm yields the smallest cost.
     $\rightarrow$ e.g., $(R \bowtie T) \bowtie U$ through a one-pass join
  3. Repeat the previous step until we have a complete join ordering.

Cost-Based Plan Selection

Greedy plan selection

- This is a generalization of the greedy algorithm to compute a join ordering described in section 16.6.6 from the book. However, we use I/O operations (in blocks) as our cost metric instead of the size of the intermediate results (in tuples) as done in the book.
- Often, the leaves of the logical query plan are selections. We have seen two physical operators for selections: table-scan and index-scan. The book describes in section 16.7.1 how we can choose the best selection method when the selection condition is complex.

Cost-Based Plan Selection

Materializing vs Pipelining

- A pipelined operator consumes some memory buffers
- A materialized operator frees all buffers consumed by descendant operators.

Question

Construct the optimal physical query plan for:

$$\pi_{\texttt{D.did},\texttt{COUNT}(*)}\gamma_{\texttt{D.did},\texttt{COUNT}(*)}(\pi_{D.did,D.pid}(\sigma_{\texttt{D.budget}\geq 99000}(D)) \bowtie \pi_{P.pid}(P))$$

To solve this, you need to know what the cost formula is for evaluating $\gamma_{\texttt{D.did},\texttt{COUNT}(*)}$.

Any ideas?