Database Systems Architecture

Stijn Vansummeren

General Course Information

Objective:

To obtain insight into the internal operation and implementation of systems designed to manage and process large amounts of data ("database management systems").

- Storage management
- Query processing
- Transaction management

General Course Information

Examples of data management systems:

- Relational DBMSs
- NoSQL DBMS
- Graph databases
- Stream processing systems/Complex event processing systems
- Distributed compute engines, e.g. Spark, Flink, . . . (to some extent)

Focus on relational DBMS, with discussion on how the foundational ideas of relational DBMSs are modified in other systems

General Course Information

Why is this interesting?

- Understand how typical data management systems work
- Predict data management system behavior, tune its performance
- Many of the techniques studied transfer to settings other than data management systems
  (MMORPGs, Financial market analysis, distributed computation, . . .)

What this course is not:

- Introduction to databases
- Focused on particular DBMS (Oracle, IBM,. . .)

General Course Information

Organisation

- Combination of lectures; exercise sessions; guided self-study; and project work.
- Evaluation: project and written exam

Course material

- Database Systems: The Complete Book (H. Garcia-Molina, J. D. Ullman, and J. Widom) second edition
- Course notes (available on website)

Contact information

- Email: `stijn.vansummeren@ulb.ac.be`
- Office: UB4.125
- Website: `http://cs.ulb.ac.be/public/teaching/infoh417`

Course Prerequisites

An introductory course on relational database systems

- Understanding of the Relational Algebra
- Understanding of SQL

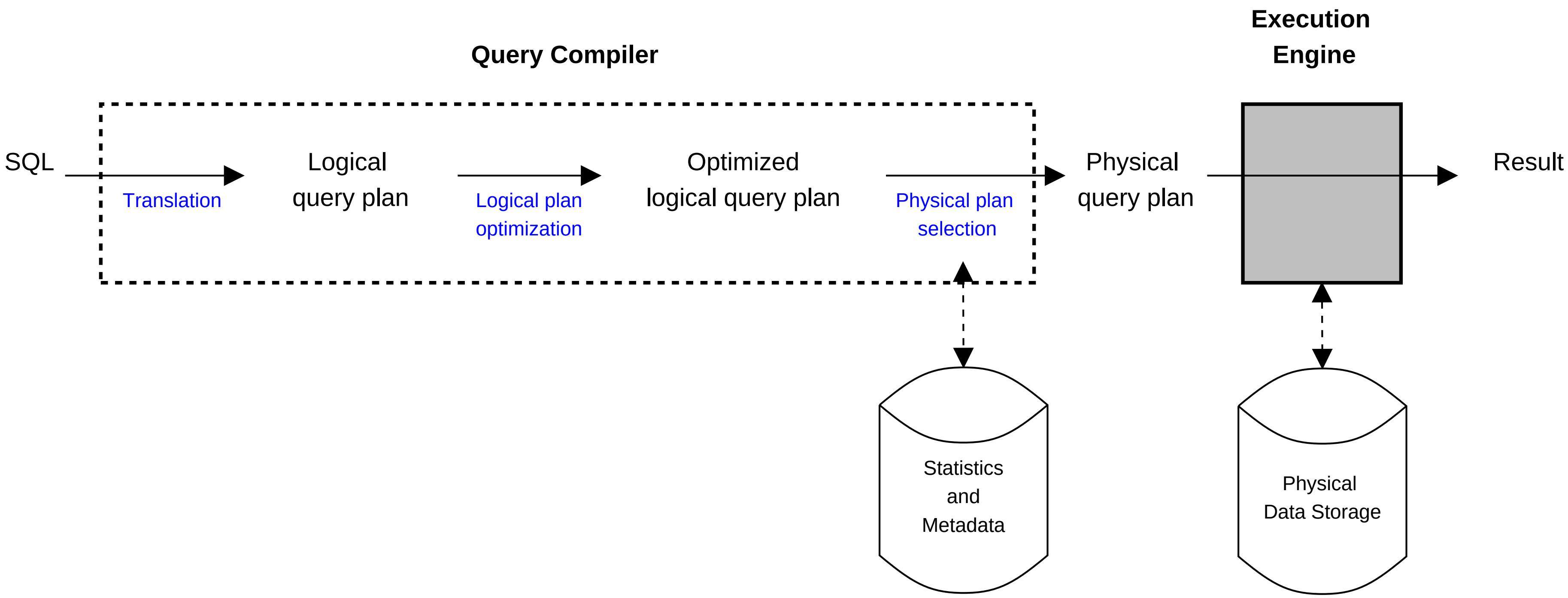
Background on basic data structures and algorithms

- Search trees
- Hashing
- Analysis of algorithms: worst-case complexity and big-oh notation (e.g., $O(n^3)$)
- Basic knowledge of what it means to be NP-complete

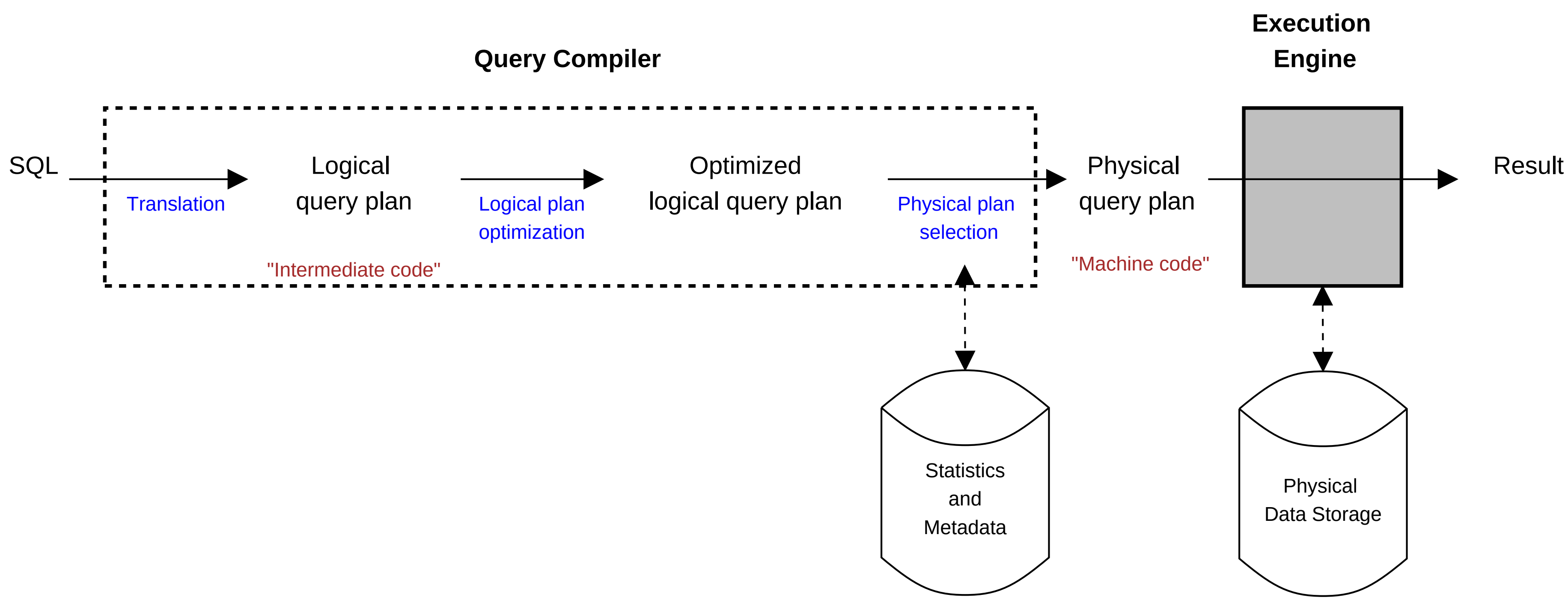
Proficiency in Programming (Java or C/C++)

- Necessary for completing the project assignment

Query processing: overview



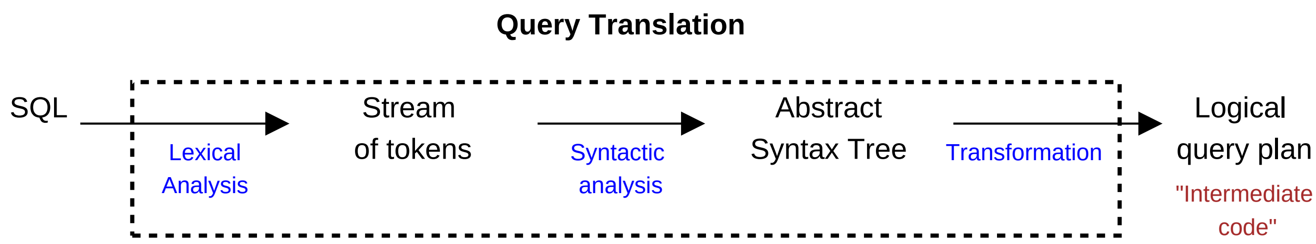
Query processing: overview



Translation of SQL into Relational Algebra

From SQL text to logical query plans

Translation of SQL into relational algebra: overview



We will adopt the following simplifying assumptions:

We will only show how to translate SQL-92 queries

And we adopt a set-based semantics of SQL. (In contrast, real SQL is bag-based.)

What will we use as logical query plans?

The extended relational algebra (interpreted over sets).

Prerequisites

- SQL: see chapter 6 in TCB
- Extended relational algebra: chapter 5 in TCB

Refreshing the Relational Algebra

Relations are tables whose columns have names, called attributes

A	B	C	D
1	2	3	4
1	2	3	5
3	4	5	6
5	6	3	4

The set of all attributes of a relation is called the schema of the relation.

The rows in a relation are called tuples.

A relation is set-based if it does not contain duplicate tuples. It is called bag-based otherwise.

Refreshing the Relational Algebra

Unless specified otherwise, we assume that relations are set-based.

Each Relational Algebra operator takes as input 1 or more relations, and produces a new relation.

Refreshing the Relational Algebra

Union (set-based)

$$
\begin{array}{c|c} A & B \\ \hline\hline 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{array}
\quad \cup \quad
\begin{array}{c|c} A & B \\ \hline\hline 3 & 4 \\ 1 & 5 \end{array}
\quad = \quad
\begin{array}{c|c} A & B \\ \hline\hline 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 1 & 5 \end{array}
$$

Input relations must have the same schema (same set of attributes)

Refreshing the Relational Algebra

Intersection (set-based)

$$\begin{array}{c|c} A & B \\ \hline\hline 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{array} \quad \cap \quad \begin{array}{c|c} A & B \\ \hline\hline 3 & 4 \\ 1 & 5 \end{array} \quad = \quad \begin{array}{c|c} A & B \\ \hline\hline 3 & 4 \end{array}$$

Input relations must have same set of attributes

Refreshing the Relational Algebra

Difference (set-based)

$$\begin{array}{c|c} A & B \\ \hline\hline 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{array} \quad - \quad \begin{array}{c|c} A & B \\ \hline\hline 3 & 4 \\ 1 & 5 \end{array} \quad = \quad \begin{array}{c|c} A & B \\ \hline\hline 1 & 2 \\ 5 & 6 \end{array}$$

Input relations must have same set of attributes

Refreshing the Relational Algebra

Selection

$$\sigma_{A>=3}\left(\begin{array}{c|c} A & B \\ \hline\hline 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{array}\right) \quad = \quad \begin{array}{c|c} A & B \\ \hline\hline 3 & 4 \\ 5 & 6 \end{array}$$

Refreshing the Relational Algebra

Projection (set-based)

$$\pi_{A,C}\left(\begin{array}{c|c|c|c} A & B & C & D \\ \hline\hline 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 5 \\ 3 & 4 & 5 & 6 \\ 5 & 6 & 3 & 4 \end{array}\right) = \begin{array}{c|c} A & C \\ \hline\hline 1 & 3 \\ 3 & 5 \\ 5 & 3 \end{array}$$

Refreshing the Relational Algebra

Cartesian product

A	B
1	2
3	4

$\times$

C	D
2	6
3	7
4	9

$=$

A	B	C	D
1	2	2	6
1	2	3	7
1	2	4	9
3	4	2	6
3	4	3	7
3	4	4	9

Input relations must have disjoint schema (set of attributes)

Refreshing the Relational Algebra

Natural Join

$$
\begin{array}{c|c} A & B \\ \hline\hline 1 & 2 \\ 3 & 4 \end{array}
\quad \bowtie \quad
\begin{array}{c|c} B & D \\ \hline\hline 2 & 6 \\ 3 & 7 \\ 4 & 9 \end{array}
\quad = \quad
\begin{array}{c|c|c} A & B & D \\ \hline\hline 1 & 2 & 6 \\ 3 & 4 & 9 \end{array}
$$

Refreshing the Relational Algebra

Natural Join

A	B
1	2
3	4

$\bowtie$

C	D
2	6
3	7
4	9

$=$

A	B	C	D
1	2	2	6
1	2	3	7
1	2	4	9
3	4	2	6
3	4	3	7
3	4	4	9

Refreshing the Relational Algebra

Theta Join

$$\begin{array}{c|c} A & B \\ \hline\hline 1 & 2 \\ 3 & 4 \end{array} \quad \bowtie_{B=C} \quad \begin{array}{c|c} C & D \\ \hline\hline 2 & 6 \\ 3 & 7 \\ 4 & 9 \end{array} \quad = \quad \begin{array}{c|c|c|c} A & B & C & D \\ \hline\hline 1 & 2 & 2 & 6 \\ 3 & 4 & 4 & 9 \end{array}$$

Refreshing the Relational Algebra

Renaming

$$\rho_T\left(\begin{array}{c|c} A & B \\ \hline\hline 1 & 2 \\ 3 & 4 \end{array}\right) \quad = \quad \begin{array}{l|l} T.A & T.B \\ \hline\hline 1 & 2 \\ 3 & 4 \end{array}$$

Renaming specifies that the input relation (and its attributes) should be given a new name.

Refreshing the Relational Algebra

Relational algebra expressions:

- Built using relation variables
- And relational algebra operators

$$\sigma_{\texttt{length} \geq 100}(\texttt{Movie}) \bowtie_{\texttt{title=movietitle}} \texttt{StarsIn}$$

Refreshing the Relational Algebra

The extended relational algebra

Adds some operators to the algebra (sorting, grouping, ...) and extends others (projection).

Grouping:

$$\gamma_{A,\min(B)\to D}\left(\begin{array}{c|c|c} A & B & C \\ \hline\hline 1 & 2 & a \\ 1 & 3 & b \\ 2 & 3 & c \\ 2 & 4 & a \\ 2 & 5 & a \end{array}\right) = \begin{array}{c|c} A & D \\ \hline\hline 1 & 2 \\ 2 & 3 \end{array}$$

Refreshing the Relational Algebra

The extended relational algebra

Adds some operators to the algebra (sorting, grouping, ...) and extends others (projection).

Extend projection to allow renaming of attributes:

$$\pi_{A,C\to D}\left(\begin{array}{c|c|c|c} A & B & C & D \\ \hline\hline 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 5 \\ 3 & 4 & 5 & 6 \\ 5 & 6 & 3 & 4 \end{array}\right) \quad = \quad \begin{array}{c|c} A & D \\ \hline\hline 1 & 3 \\ 3 & 5 \\ 5 & 3 \end{array}$$

Refreshing the Relational Algebra

On the difference between sets and bags

- Historically speaking, relations are defined to be sets of tuples: duplicate tuples cannot occur in a relation.
- In practical systems, however, it is more efficient to allow duplicates to occur in relations, and only remove duplicates when requested. In this case relations are bags.

Union (bag-based)

$$\begin{array}{c|c} A & B \\ \hline\hline 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{array} \quad \cup \quad \begin{array}{c|c} A & B \\ \hline\hline 3 & 4 \\ 1 & 5 \end{array} \quad = \quad \begin{array}{c|c} A & B \\ \hline\hline 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 3 & 4 \\ 1 & 5 \end{array}$$

Refreshing the Relational Algebra

On the difference between sets and bags

- Historically speaking, relations are defined to be sets of tuples: duplicate tuples cannot occur in a relation.
- In practical systems, however, it is more efficient to allow duplicates to occur in relations, and only remove duplicates when requested. In this case relations are bags.

Intersection (bag-based)

A	B
1	2
3	4
1	2
1	2

$\cap$

A	B
1	2
3	4
3	4
5	6

$=$

A	B
1	2
3	4

Refreshing the Relational Algebra

On the difference between sets and bags

- Historically speaking, relations are defined to be sets of tuples: duplicate tuples cannot occur in a relation.
- In practical systems, however, it is more efficient to allow duplicates to occur in relations, and only remove duplicates when requested. In this case relations are bags.

Difference (bag-based)

$$\begin{array}{c|c} A & B \\ \hline\hline 1 & 2 \\ 3 & 4 \\ 1 & 2 \\ 1 & 2 \end{array} \quad - \quad \begin{array}{c|c} A & B \\ \hline\hline 1 & 2 \\ 3 & 4 \\ 3 & 4 \\ 5 & 6 \end{array} \quad = \quad \begin{array}{c|c} A & B \\ \hline\hline 1 & 2 \\ 1 & 2 \end{array}$$

Refreshing the Relational Algebra

On the difference between sets and bags

- Historically speaking, relations are defined to be sets of tuples: duplicate tuples cannot occur in a relation.
- In practical systems, however, it is more efficient to allow duplicates to occur in relations, and only remove duplicates when requested. In this case relations are bags.

Projection (bag-based)

$$\pi_{A,C}\left(\begin{array}{c|c|c|c} A & B & C & D \\ \hline\hline 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 5 \\ 3 & 4 & 5 & 6 \\ 5 & 6 & 3 & 4 \end{array}\right) \quad = \quad \begin{array}{c|c} A & C \\ \hline\hline 1 & 3 \\ 1 & 3 \\ 3 & 5 \\ 5 & 3 \end{array}$$

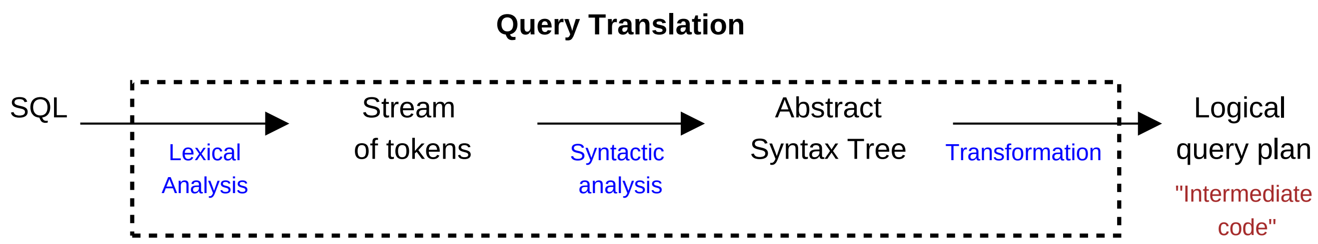
Refreshing the Relational Algebra

On the difference between sets and bags

- Historically speaking, relations are defined to be sets of tuples: duplicate tuples cannot occur in a relation.
- In practical systems, however, it is more efficient to allow duplicates to occur in relations, and only remove duplicates when requested. In this case relations are bags.

The other operators are straightforwardly extended to bags: simply do the same operation, taking into account duplicates

Translation of SQL into relational algebra: overview



We will adopt the following simplifying assumptions:

We will only show how to translate SQL-92 queries

And we adopt a set-based semantics of SQL. (In contrast, real SQL is bag-based.)

What will we use as logical query plans?

The extended relational algebra (interpreted over sets).

Prerequisites

- SQL: see chapter 6 in TCB
- Extended relational algebra: chapter 5 in TCB

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: ???

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: $\boldsymbol{\pi}_{\texttt{movieTitle}}\, \boldsymbol{\sigma}_{\substack{\texttt{S.starName}=\texttt{M.name} \\ \wedge \texttt{M.birthdate}=1960}} (\boldsymbol{\rho}_{\texttt{S}}(\texttt{StarsIn}) \times \boldsymbol{\rho}_{\texttt{M}}(\texttt{MovieStar}))$

Translation of SQL into the relational algebra

Select statements in general contain subqueries

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

Subqueries in the where-clause

Occur through the operators =, <, >, <=, >=, <>; through the quantifiers `ANY`, or `ALL`; or through the operators `EXISTS` and `IN` and their negations `NOT EXISTS` and `NOT IN`.

Translation of SQL into the relational algebra

We can always normalize subqueries to use only `EXISTS` **and** `NOT EXISTS`

```
SELECT movieTitle FROM StarsIn
WHERE starName IN (SELECT name
                   FROM MovieStar
                   WHERE birthdate=1960)
```

$\Rightarrow$
```
SELECT movieTitle FROM StarsIn
WHERE EXISTS (SELECT name
              FROM MovieStar
              WHERE birthdate=1960 AND name=starName)
```

Translation of SQL into the relational algebra

We can always normalize subqueries to use only `EXISTS` and `NOT EXISTS`

```
SELECT name FROM MovieExec
WHERE netWorth >= ALL (SELECT E.netWorth
                       FROM MovieExec E)
```

```
⇒ SELECT name FROM MovieExec
   WHERE NOT EXISTS(SELECT E.netWorth
                    FROM MovieExec E
                    WHERE netWorth < E.netWorth)
```

Translation of SQL into the relational algebra

We can always normalize subqueries to use only `EXISTS` **and** `NOT EXISTS`

```
SELECT C FROM S
WHERE C IN (SELECT SUM(B) FROM R
            GROUP BY A)
```

$\Rightarrow$???

Translation of SQL into the relational algebra

We can always normalize subqueries to use only `EXISTS` and `NOT EXISTS`

```
SELECT C FROM S
WHERE C IN (SELECT SUM(B) FROM R
            GROUP BY A)
```

$\Rightarrow$
```
SELECT C FROM S
WHERE EXISTS (SELECT SUM(B) FROM R
              GROUP BY A
              HAVING SUM(B) = C)
```

Translation of SQL into the relational algebra

Translating subqueries - First step: normalization

- Before translating a query we first normalize it such that all of the subqueries that occur in a `WHERE` condition are of the form `EXISTS` or `NOT EXISTS`.
- We may hence assume without loss of generality in what follows that all subqueries in a `WHERE` condition are of the form `EXISTS` or `NOT EXISTS`.

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.

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

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.

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

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

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

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.

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

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 $\boldsymbol{\rho}_S(\texttt{StarsIn})$

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

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)
```

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

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

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.

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

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 NOT EXISTS (SELECT name
                FROM MovieStar
                WHERE birthdate=1960 AND name= S.starName)
```

1. We first translate the `NOT EXISTS` subquery.

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

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

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

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.

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

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

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

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

Translation of SQL into the relational algebra

Translation of correlated select-from-where subqueries

In the previous examples we have only considered queries of the following form:

SELECT Select-list FROM From-list
WHERE ψ AND EXISTS(Q) AND $\cdots$ AND NOT EXISTS(P) AND $\cdots$

How do we treat the following?

SELECT Select-list FROM From-list
WHERE A=B AND NOT(EXISTS(Q) AND C<6)

Translation of SQL into the relational algebra

Translation of correlated select-from-where subqueries

In the previous examples we have only considered queries of the following form:

```
SELECT Select-list FROM From-list
WHERE ψ AND EXISTS(Q) AND ··· AND NOT EXISTS(P) AND ···
```

How do we treat the following?

```
SELECT Select-list FROM From-list
WHERE A=B AND NOT(EXISTS(Q) AND C<6)
```

1. We first transform the condition into disjunctive normal form:

```
SELECT Select-list FROM From-list
WHERE (A=B AND NOT EXISTS(Q)) OR (A=B AND C>=6)
```

Translation of SQL into the relational algebra

Translation of correlated select-from-where subqueries

In the previous examples we have only considered queries of the following form:

SELECT Select-list FROM From-list
WHERE ψ AND EXISTS(Q) AND $\cdots$ AND NOT EXISTS(P) AND $\cdots$

How do we treat the following?

```
SELECT Select-list FROM From-list
WHERE A=B AND NOT(EXISTS(Q) AND C<6)
```

2. We then distribute the OR

```
(SELECT Select-list FROM From-list
WHERE (A=B AND NOT EXISTS(Q)))
UNION
(SELECT Select-list FROM From-list
WHERE (A=B AND C>=6))
```

Translation of SQL into the relational algebra

Union, intersection, and difference

SQL: `(SELECT * FROM R R1) INTERSECT (SELECT * FROM R R2)`

Algebra: $\boldsymbol{\rho}_{R_1}(R) \cap \boldsymbol{\rho}_{R_2}(R)$

SQL: `(SELECT * FROM R R1) UNION (SELECT * FROM R R2)`

Algebra: $\boldsymbol{\rho}_{R_1}(R) \cup \boldsymbol{\rho}_{R_2}(R)$

SQL: `(SELECT * FROM R R1) EXCEPT (SELECT * FROM R R2)`

Algebra: $\boldsymbol{\rho}_{R_1}(R) - \boldsymbol{\rho}_{R_2}(R)$

Translation of SQL into the relational algebra

Union, intersection, and difference in subqueries

Consider the relations $R(A, B)$ and $S(C)$.

```
SELECT S1.C, S2.C
FROM S S1, S S2
WHERE EXISTS (
  (SELECT R1.A, R1.B FROM R R1
   WHERE A = S1.C AND B = S2.C)
 UNION
  (SELECT R2.A, R2.B FROM R R2
   WHERE B = S1.C)
)
```

In this case we translate the subquery as follows:

$$\boldsymbol{\pi}_{S_1.C,S_2.C,R_1.A\to A,R_1.B\to B}\,\boldsymbol{\sigma}_{\substack{A=S_1.C\\ \wedge B=S_2.C}}\,(\boldsymbol{\rho}_{R_1}(R)\times\boldsymbol{\rho}_{S_1}(S)\times\boldsymbol{\rho}_{S_2}(S))$$
$$\cup\,\boldsymbol{\pi}_{S_1.C,S_2.C,R_2.A\to A,R_2.B\to B}\,\boldsymbol{\sigma}_{B=S_1.C}\,(\boldsymbol{\rho}_{R_2}(R)\times\boldsymbol{\rho}_{S_1}(S)\times\boldsymbol{\rho}_{S_2}(S))$$

Translation of SQL into the relational algebra

Join-expressions

SQL: `(SELECT * FROM R R1) CROSS JOIN (SELECT * FROM R R2)`

Algebra: $\boldsymbol{\rho}_{R_1}(R) \times \boldsymbol{\rho}_{R_2}(R)$

SQL:
```
(SELECT * FROM R R1) JOIN (SELECT * FROM R R2)
  ON R1.A = R2.B
```

Algebra: $\boldsymbol{\rho}_{R_1}(R) \underset{R_1.A=R_2.B}{\bowtie} \boldsymbol{\rho}_{R_2}(R)$

Translation of SQL into the relational algebra

Join-expressions in subqueries

Consider the relations $R(A, B)$ and $S(C)$.

```
SELECT S1.C, S2.C
FROM S S1, S S2
WHERE EXISTS (
  (SELECT R1.A, R1.B FROM R R1
   WHERE A = S1.C AND B = S2.C)
  CROSS JOIN
  (SELECT R2.A, R2.B FROM R R2
   WHERE B = S1.C)
)
```

In this case we translate the subquery as follows:

$$\boldsymbol{\pi}_{S_1.C, S_2.C, R_1.A, R_1.B}\, \boldsymbol{\sigma}_{\substack{A = S_1.C \\ \wedge B = S_2.C}} \left(\boldsymbol{\rho}_{R_1}(R) \times \boldsymbol{\rho}_{S_1}(S) \times \boldsymbol{\rho}_{S_2}(S)\right)$$

$$\bowtie \boldsymbol{\pi}_{S_1.C, R_2.A, R_2.B}\, \boldsymbol{\sigma}_{B = S_1.C} \left(\boldsymbol{\rho}_{R_2}(R) \times \boldsymbol{\rho}_{S_1}(S)\right)$$

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:

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

Translation of SQL into the relational algebra

Subqueries in the From-list

SQL:

```
SELECT movieTitle
FROM StarsIn, (SELECT name FROM MovieStar
                      WHERE birthdate = 1960) M
WHERE starName = M.name
```

Algebra:

$$\boldsymbol{\pi}_{\texttt{movieTitle}}\boldsymbol{\sigma}_{\texttt{starName}=\texttt{M.name}}\big(\texttt{StarsIn} \times \boldsymbol{\rho}_{\texttt{M}}\boldsymbol{\pi}_{\texttt{name}}\boldsymbol{\sigma}_{\texttt{birthdate}=1960}(\texttt{MovieStar})\big)$$

Translation of SQL into the relational algebra

Lateral subqueries in SQL-99

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

1. We first translate the first subquery

$$E_1 = \boldsymbol{\pi}_{\texttt{name}}\, \boldsymbol{\sigma}_{\texttt{birthdate}=1960}(\texttt{MovieStar}).$$

2. We then translate the second subquery, which has E_1 as context relation:

$$E_2 = \rho_S \boldsymbol{\pi}_{\texttt{name},\texttt{movieTitle}}\, \boldsymbol{\sigma}_{\texttt{starName}=\texttt{M.name}}(\texttt{StarsIn} \times E_1).$$

3. Finally, we translate the whole `FROM`-clause by means of a join due to the correlation:

$$\boldsymbol{\pi}_{\texttt{movieTitle}}(E_1 \bowtie E_2).$$

Translation of SQL into the relational algebra

Lateral subqueries in SQL-99

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

4. In this example, however, all relevant tuples of E_1 are already contained in the result of E_2, and we can hence simplify:

$$\boldsymbol{\pi}_{\texttt{movieTitle}}(E_2).$$

Translation of SQL into the relational algebra

Subqueries in the select-list

Consider again the relations $R(A, B)$ and $S(C)$, and assume that A is a key for R. The following query is then permitted:

```
SELECT C, (SELECT B FROM R
           WHERE A=C)
FROM S
```

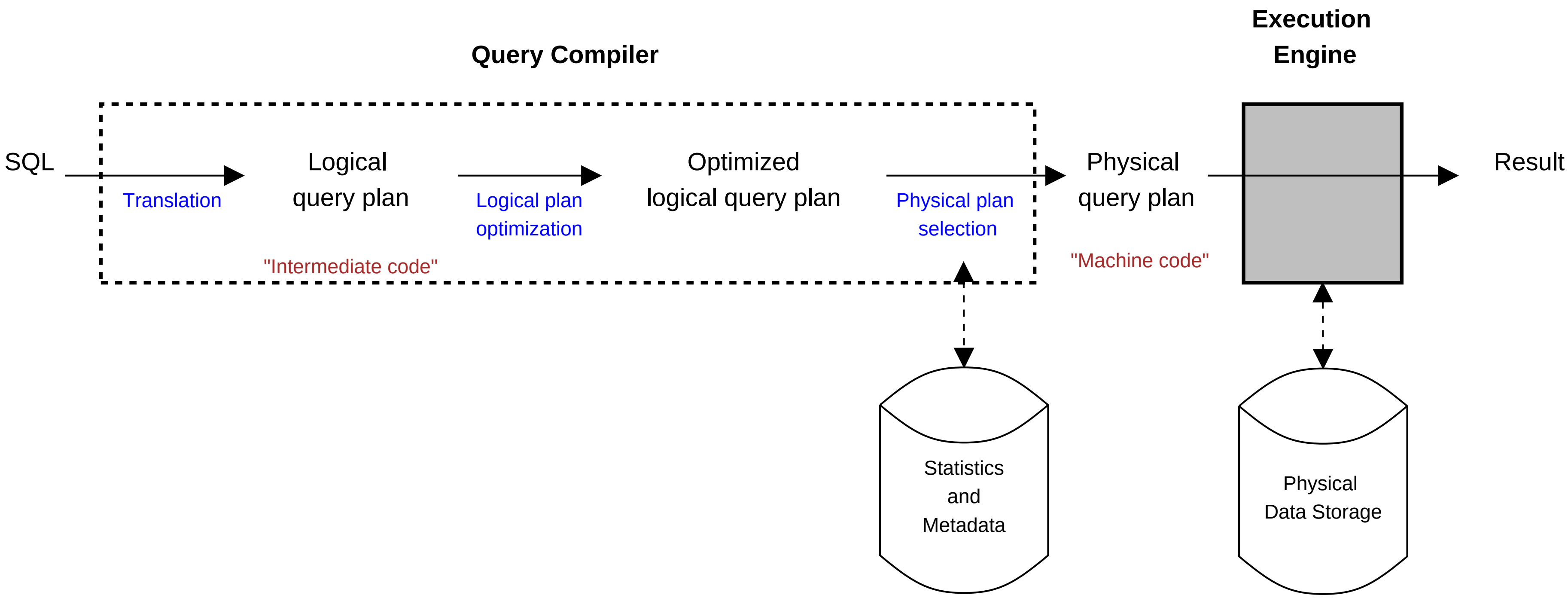
Such queries can be rewritten as queries with `LATERAL` subqueries in the from-list:

```
SELECT C, T.B
FROM (SELECT C FROM S),
     LATERAL
     (SELECT B FROM R
      WHERE A=C) T
```

We can hence first rewrite them in `LATERAL` form, and subsequently translate the rewritten query into the relational algebra.

Optimization of logical query plans
Eliminating redundant joins

Optimization of logical query plans



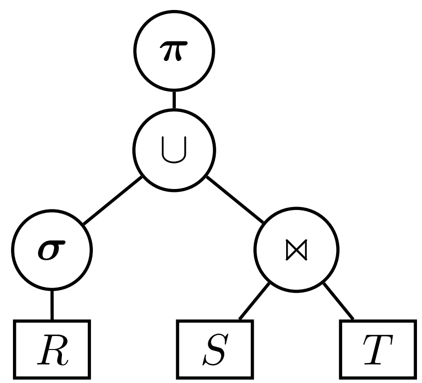
Optimization of logical query plans

Logical plans = execution trees

A logical query plan like

$$\pi_{A,B}(\sigma_{A=5}(R) \cup (S \bowtie T))$$

is essentially an execution tree



A physical query plan is a logical query plan in which each node is assigned the algorithm that should be used to evaluate the corresponding relational algebra operator. (There are multiple ways to evaluate each algebra operator)

Optimization of logical query plans

The need to optimize

- Every internal node (i.e., each occurrence of an operator) in the plan must be executed.
- Hence, the fewer nodes we have, the faster the execution

Definition

A relational algebra expression e is optimal if there is no other expression e' that is both (1) equivalent to e and (2) "shorter" than e (i.e., has fewer occurrence of operators than e).

The optimization problem:

Input: a relational algebra expression e

Output: an optimal relational algebra expression e' equivalent to e.

Optimization of logical query plans

The optimization problem is undecidable:

It is known that the following problem is undecidable:

Input: relational algebra expressions e_1 and e_2 over a single relation $R(A, B)$
Output: $e_1 \equiv e_2$?

Proof: Suppose that we can compute e', the optimal expression for $(e_1 - e_2) \cup (e_2 - e_1)$. Note that $e_1 \equiv e_2$ if, and only if, e' is either $\sigma_{\texttt{false}}(R)$ or $R - R$.

Conclusion: the optimization problem is undecidable

Question: can we optimize plans that are of a particular form?

Optimization of select-project-join expressions

In practice, most SQL queries are of the following form:

```
SELECT ...
FROM R1, R2, ..., Rm
WHERE A1 = B1 AND A2 = B2 AND ...  AND An = Bn
```

The corresponding logical query plans are of the form:

$$\boldsymbol{\pi}_{\ldots}\ \boldsymbol{\sigma}_{A_1=B_1 \wedge A_2=B_2 \wedge \cdots \wedge A_n=B_n}\ (R_1 \times R_2 \times \cdots \times R_m).$$

We call such relational algebra expressions select-project-join expressions (SPJ expressions for short)

Optimization of select-project-join expressions

Removing redundant joins:

A careless SQL programmer writes the following query:

```
SELECT movieTitle FROM StarsIn S1
WHERE starName IN (SELECT name
                   FROM MovieStar, StarsIn S2
                   WHERE birthdate = 1960
                    AND S2.movieTitle = S1.movieTitle)
```

This query is equivalent to the following one, which has one join less to execute!

```
SELECT movieTitle FROM StarsIn
WHERE starName IN (SELECT name
                   FROM MovieStar
                   WHERE birthdate = 1960)
```

Optimization of select-project-join expressions

Here are the corresponding logical query plans:

$$\boldsymbol{\pi}_{S_1.\texttt{movieTitle}}(\boldsymbol{\rho}_{S_2}(\texttt{StarsIn}) \underset{S_2.\texttt{movieTitle}=S_1.\texttt{movieTitle}}{\bowtie} \boldsymbol{\rho}_{S_1}(\texttt{StarsIn}) \underset{S_1.\texttt{starName}=\texttt{name}}{\bowtie} \boldsymbol{\sigma}_{\texttt{birthdate}=1960}(\texttt{MovieStar}))$$

versus

$$\boldsymbol{\pi}_{S_1.\texttt{movieTitle}}(\boldsymbol{\rho}_{S_1}(\texttt{StarsIn}) \underset{S_1.\texttt{starName}=\texttt{name}}{\bowtie} \boldsymbol{\sigma}_{\texttt{birthdate}=1960}(\texttt{MovieStar})).$$

Redundant joins may also be introduced because of view expansion

Can we optimize select-project-join expressions? (And hence remove redundant joins?)

Optimization of select-project-join expressions

Definition

A conjunctive query is an expression of the form

$$Q\underbrace{(x_1, \ldots, x_n)}_{\text{head}} \leftarrow \underbrace{R(t_1, \ldots, t_m), \ldots, S(t'_1, \ldots, t'_k)}_{\text{body}}$$

Here $t_1, \ldots, t'_k$ denote variables and constants, and $x_1, \ldots, x_n$ must be variables that occur in $t_1, \ldots, t'_k$. We call an expression like $R(t_1, \ldots, t_m)$ an atom. If an atom does not contain any variables, and hence consists solely of contstants, then it is called a fact.

Optimization of select-project-join expressions

Semantics of conjunctive queries

Consider the following toy database D:

$$
\begin{array}{c}
R \\
\boxed{\begin{array}{cc} 1 & 2 \\ 2 & 3 \\ 2 & 5 \\ 6 & 7 \\ 7 & 5 \\ 5 & 5 \end{array}}
\end{array}
\qquad
\begin{array}{c}
S \\
\boxed{\begin{array}{c} 2 \\ 7 \end{array}}
\end{array}
$$

as well as the following conjunctive query over the relations $R(A, B)$ and $S(C)$:

$$Q(x, y) \leftarrow R(x, y),\ R(y, 5),\ S(y).$$

Intuitively, Q wants to retrieve all pairs of values (x, y) such that (1) this pair occurs in relation R; (2) y occurs together with the constant 5 in a tuple in R; and (3) y occurs as a value in S. The formal definition is as follows.

Optimization of select-project-join expressions

Semantics of conjunctive queries

Consider the following toy database D:

R	
1	2
2	3
2	5
6	7
7	5
5	5

S
2
7

as well as the following conjunctive query over the relations $R(A, B)$ and $S(C)$:

$$Q(x, y) \leftarrow R(x, y),\ R(y, 5),\ S(y).$$

A substitution f of Q into D is a function that maps variables in Q to constants in D. For example:

$$f\colon \begin{array}{ccc} x & \mapsto & 1 \\ y & & 2 \end{array}$$

Optimization of select-project-join expressions

Semantics of conjunctive queries

Consider the following toy database D:

R	
1	2
2	3
2	5
6	7
7	5
5	5

S
2
7

as well as the following conjunctive query over the relations $R(A, B)$ and $S(C)$:

$$Q(x, y) \leftarrow R(x, y),\ R(y, 5),\ S(y).$$

A matching is a substitution that maps the body of Q into facts in D. For example:

$$f\colon \begin{array}{ccc} x & \mapsto & 1 \\ y & & 2 \end{array}$$

Optimization of select-project-join expressions

Semantics of conjunctive queries

Consider the following toy database D:

R		S
1	2	2
2	3	7
2	5	
6	7	
7	5	
5	5	

as well as the following conjunctive query over the relations $R(A, B)$ and $S(C)$:

$$Q(x, y) \leftarrow R(x, y),\ R(y, 5),\ S(y).$$

The result of a conjunctive query is obtained by applying all possible matchings to the head of the query. In our example:

$$Q(D) = \{(1, 2), (6, 7)\}.$$

Optimization of select-project-join expressions

Translation of SPJ expressions into conjunctive queries

Schema:

- 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`)

SPJ: $\boldsymbol{\pi}_{\texttt{title}}(\texttt{Movie} \underset{\texttt{title}=\texttt{movieTitle}}{\bowtie} \texttt{StarsIn} \underset{\texttt{starName}=\texttt{name}}{\bowtie} \boldsymbol{\sigma}_{\texttt{birthdate}=1960}(\texttt{MovieStar}))$

CQ: $Q_1(t) \leftarrow \texttt{Movie}(t, y, \ell, i, s, p), \texttt{StarsIn}(t, y_2, n), \texttt{MovieStar}(n, a, g, 1960)$

Optimization of select-project-join expressions

Translation of conjunctive queries into SPJ expressions

<u>Schema:</u>

- R(A,B)
- S(C)

CQ: $Q(x,y) \leftarrow R(x,y),\ R(y,5),\ S(y)$

SPJ: $\boldsymbol{\pi}_{R_1.A,R_1.B}\,\boldsymbol{\sigma}_{R_1.B=R_2.A}\,\boldsymbol{\sigma}_{R_2.B=5}\,\boldsymbol{\sigma}_{R_1.B=C}(\boldsymbol{\rho}_{R_1}(R) \times \boldsymbol{\rho}_{R_2}(R) \times S)$

Optimization of select-project-join expressions

Translation of SPJ expressions into conjunctive queries

SPJ: $\boldsymbol{\pi}_{\texttt{title}}(\texttt{Movie} \underset{\texttt{title=movieTitle}}{\bowtie} \texttt{StarsIn} \underset{\texttt{starName=name}}{\bowtie} \boldsymbol{\sigma}_{\texttt{birthdate}=1960}(\texttt{MovieStar}))$

CQ: $Q_1(t) \leftarrow \texttt{Movie}(t, y, \ell, i, s, p),\ \texttt{StarsIn}(t, y_2, n),\ \texttt{MovieStar}(n, a, g, 1960)$

Translation of conjunctive queries into SPJ expressions

CQ: $Q(x, y) \leftarrow R(x, y),\ R(y, 5),\ S(y)$

SPJ: $\boldsymbol{\pi}_{R_1.A, R_1.B}\, \boldsymbol{\sigma}_{R_1.B=R_2.A}\, \boldsymbol{\sigma}_{R_2.B=5}\, \boldsymbol{\sigma}_{R_1.B=C}(\boldsymbol{\rho}_{R_1}(R) \times \boldsymbol{\rho}_{R_2}(R) \times S)$

Conclusion:

Select-project-join expressions and conjunctive queries are two separate syntaxes for the same class of queries.

Optimization of select-project-join expressions

In-class exercise

Consider the following SQL expression over the relations $R(A, B)$ and $S(B, C)$:

```
SELECT R1.A, S1.B
FROM R R1, R R2, R R3, S S1, S S2
WHERE
    R1.A = R2.A AND R2.B =4 AND R2.A = R3.A
AND R3.B = S1.B AND S1.C = S2.C AND S2.B=4
```

Exercise: Translate this SQL expression into a logical query plan. If this plan is an SPJ-expression, then translate this plan into a conjunctive query.

Optimization of select-project-join expressions

In-class exercise

Consider the following SQL expression over the relations $R(A, B)$ and $S(B, C)$:

```
SELECT R1.A, S1.B
FROM R R1, R R2, R R3, S S1, S S2
WHERE
    R1.A = R2.A AND R2.B =4 AND R2.A = R3.A
AND R3.B = S1.B AND S1.C = S2.C AND S2.B=4
```

Here is the logical query plan:

$$\pi_{R_1.A,S_1.B}\sigma_{R_1.A=R_2.A\wedge R_2.B=4\wedge R_2.A=R_3.A\wedge R_3.B=S_1.B\wedge S_1.C=S_2.C\wedge S_2.B=4}$$
$$(\rho_{R_1}(R)\times\rho_{R_2}(R)\times\rho_{R_3}(R)\times\rho_{S_1}(S)\times\rho_{S_2}(S))$$

Optimization of select-project-join expressions

In-class exercise

Consider the following SQL expression over the relations $R(A, B)$ and $S(B, C)$:

```
SELECT R1.A, S1.B
FROM R R1, R R2, R R3, S S1, S S2
WHERE
    R1.A = R2.A AND R2.B =4 AND R2.A = R3.A
AND R3.B = S1.B AND S1.C = S2.C AND S2.B=4
```

Here is the logical query plan:

$$\pi_{R_1.A,S_1.B}\sigma_{R_1.A=R_2.A\wedge R_2.B=4\wedge R_2.A=R_3.A\wedge R_3.B=S_1.B\wedge S_1.C=S_2.C\wedge S_2.B=4}$$
$$(\rho_{R_1}(R) \times \rho_{R_2}(R) \times \rho_{R_3}(R) \times \rho_{S_1}(S) \times \rho_{S_2}(S))$$

Constructing the conjunctive query: first step

$$Q(x_{R_1.A}, y_{S_1.B}) \leftarrow R(x_{R_1.A}, y_{R_1.B}), R(x_{R_2.A}, y_{R_2.B}), R(x_{R_3.A}, y_{R_3.B}),$$
$$S(y_{S_1.B}, z_{S_1.C}), S(y_{S_2.B}, z_{S_2.C})$$

Optimization of select-project-join expressions

In-class exercise

Consider the following SQL expression over the relations $R(A, B)$ and $S(B, C)$:

```
SELECT R1.A, S1.B
FROM R R1, R R2, R R3, S S1, S S2
WHERE
    R1.A = R2.A AND R2.B =4 AND R2.A = R3.A
AND R3.B = S1.B AND S1.C = S2.C AND S2.B=4
```

Here is the logical query plan:

$$\pi_{R_1.A,S_1.B}\sigma_{R_1.A=R_2.A \wedge R_2.B=4 \wedge R_2.A=R_3.A \wedge R_3.B=S_1.B \wedge S_1.C=S_2.C \wedge S_2.B=4} (\rho_{R_1}(R) \times \rho_{R_2}(R) \times \rho_{R_3}(R) \times \rho_{S_1}(S) \times \rho_{S_2}(S))$$

Constructing the conjunctive query: forcing equalities

$$Q(x_{R_1.A}, y_{S_1.B}) \leftarrow R(x_{R_1.A}, y_{R_1.B}), R(x_{R_1.A}, 4), R(x_{R_1.A}, y_{S_1.B}), S(y_{S_1.B}, z_{S_1.C}), S(4, z_{S_1.C})$$

Optimization of select-project-join expressions

In-class exercise

Consider the following SQL expression over the relations $R(A, B)$ and $S(B, C)$:

```
SELECT R1.A, S1.B
FROM R R1, R R2, R R3, S S1, S S2
WHERE
    R1.A = R2.A AND R2.B =4 AND R2.A = R3.A
AND R3.B = S1.B AND S1.C = S2.C AND S2.B=4
```

Here is the logical query plan:

$$\pi_{R_1.A,S_1.B}\sigma_{R_1.A=R_2.A\wedge R_2.B=4\wedge R_2.A=R_3.A\wedge R_3.B=S_1.B\wedge S_1.C=S_2.C\wedge S_2.B=4}$$
$$(\rho_{R_1}(R) \times \rho_{R_2}(R) \times \rho_{R_3}(R) \times \rho_{S_1}(S) \times \rho_{S_2}(S))$$

Constructing the conjunctive query: renaming variables (optional)

$$Q(x, y) \leftarrow R(x, u), R(x, 4), R(x, y), S(y, z), S(4, z)$$

Optimization of select-project-join expressions

Another in-class exercise

Consider the following conjunctive query over the relations

- MovieStar(<u>name: `string`</u>, address: `string`, gender: `char`, birthdate: `date`)
- StarsIn(<u>movieTitle: `string`, movieYear: `string`, starName: `string`</u>)

$$Q(t) \leftarrow \texttt{MovieStar}(n, a, g, 1940), \texttt{StarsIn}(t, y, n)$$

Translate this conjunctive query into an SPJ expression. What is the corresponding SQL query?

Optimization of select-project-join expressions

Another in-class exercise

Consider the following conjunctive query over the relations

- MovieStar(name: $\texttt{string}$, address: $\texttt{string}$, gender: $\texttt{char}$, birthdate: $\texttt{date}$)
- StarsIn(movieTitle: $\texttt{string}$, movieYear: $\texttt{string}$, starName: $\texttt{string}$)

$$Q(t) \leftarrow \texttt{MovieStar}(n, a, g, 1940), \texttt{StarsIn}(t, y, n)$$

Translate this conjunctive query into an SPJ expression. What is the corresponding SQL query?

SPJ expression:

$$\pi_{S.\texttt{movieTitle}}(\sigma_{M.\texttt{starName}=S.\texttt{starName}}\sigma_{M.\texttt{birthdate = 1940}}(\rho_M(\texttt{MovieStar})\times\rho_S(\texttt{StarsIn}))$$

Optimization of select-project-join expressions

Another in-class exercise

Consider the following conjunctive query over the relations

- MovieStar($\underline{\text{name: } \texttt{string}}$, address: `string`, gender: `char`, birthdate: `date`)
- StarsIn($\underline{\text{movieTitle: } \texttt{string}\text{, movieYear: } \texttt{string}\text{, starName: } \texttt{string}}$)

$$Q(t) \leftarrow \texttt{MovieStar}(n, a, g, 1940), \texttt{StarsIn}(t, y, n)$$

Translate this conjunctive query into an SPJ expression. What is the corresponding SQL query?

SPJ expression:

$\pi_{S.\texttt{movieTitle}}(\sigma_{M.\texttt{starName}=S.\texttt{starName}}\sigma_{M.\texttt{birthdate = 1940}}(\rho_M(\texttt{MovieStar})\times\rho_S(\texttt{StarsIn}))$

SQL query:

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

Optimization of select-project-join expressions

Containment of conjunctive queries

Q_1 is contained in Q_2 if $Q_1(D) \subseteq Q_2(D)$, for every database D.

Example:

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

Then B is contained in A. Proof:

1. Let D be an arbitrary database, and let $t \in B(D)$.
2. Then there exists a matching f of B into D such that $t = (f(x), f(y))$. We need to show that $(f(x), f(y)) \in A(D)$.
3. Let h be the following substitution:

$$x \rightarrow f(x) \quad y \rightarrow f(y) \quad w \rightarrow f(w) \quad z \rightarrow f(w)$$

4. Then h is a matching of A into D, and hence

$$t = (f(x), f(y)) = (h(x), h(y)) \in A(D)$$

Optimization of select-project-join expressions

Containment of conjunctive queries is decidable

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

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

1. First calculate the canonical database D for B:

$$R \quad \boxed{\begin{array}{cc} \dot{x} & \dot{w} \\ \dot{w} & \dot{y} \end{array}} \qquad G \quad \boxed{\begin{array}{cc} \dot{w} & \dot{w} \end{array}}$$

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':

$$\begin{array}{ccccc}
\mathbf{A} & \xrightarrow{h} & \mathbf{D} = \mathbf{B} & \xrightarrow{f} & \mathbf{D'} \\
\begin{array}{cc} R & G \\ \boxed{\begin{array}{cc} x & w \\ z & y \end{array}} & \boxed{\begin{array}{cc} w & z \end{array}} \end{array} & & \begin{array}{cc} R & G \\ \boxed{\begin{array}{cc} \dot{x} & \dot{w} \\ \dot{w} & \dot{y} \end{array}} & \boxed{\begin{array}{cc} \dot{w} & \dot{w} \end{array}} \end{array} & & \begin{array}{cc} R & G \\ \boxed{\begin{array}{c} \ldots \\ \ldots \\ \ldots \\ \ldots \end{array}} & \boxed{\begin{array}{c} \ldots \\ \ldots \\ \ldots \\ \ldots \end{array}} \end{array}
\end{array}$$

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':

$$\mathbf{A} \xrightarrow{h} \mathbf{D} = \mathbf{B} \xrightarrow{f} \mathbf{D'}$$

A

R

x	w
z	y

G

w	z

D = B

R

x	w
w	y

G

w	w

D′

R

...
...
...
...

G

...
...
...
...

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.

  In our example we first try to delete $R(x,x)$:

  $Q_1(x) \leftarrow R(x,y)$

  Note that $Q \subseteq Q_1$ but $Q_1 \not\subseteq Q$. We hence cannot remove this atom.

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 logical query plans

Optimization of complete, arbitrary query plans

1. Detect and optimize the select-project-join sub-expressions in the plan
2. Then use heuristics to further optimize the modified plan.

Heuristic: rewriting through algebraic laws

Consider relations $R(A, B)$ and $S(C, D)$ and the following expression that we want to optimize

$$\boldsymbol{\pi}_A \boldsymbol{\sigma}_{A=5 \land B<D}(R \times S)$$

Pushing selections:

$$\boldsymbol{\pi}_A \boldsymbol{\sigma}_{B<D}(\boldsymbol{\sigma}_{A=5}(R) \times S)$$

Recognizing joins:

$$\boldsymbol{\pi}_A(\boldsymbol{\sigma}_{A=5}(R) \underset{B<D}{\bowtie} S)$$

Introduce projections where possible:

$$\boldsymbol{\pi}_A(\boldsymbol{\sigma}_{A=5}(R) \underset{B<D}{\bowtie} \pi_D(S))$$

Optimization of logical query plans

An in-class integrated exercise

Recall the relational schema.

- MovieStar(<u>name: `string`</u>, address: `string`, gender: `char`, birthdate: `date`)
- StarsIn(<u>movieTitle: `string`, movieYear: `string`, starName: `string`</u>)

A careless SQL programmer writes the following query:

```
SELECT S1.movieTitle FROM StarsIn S1
WHERE S1.starName IN (SELECT name
                      FROM MovieStar, StarsIn S2
                      WHERE birthdate = 1960
                      AND S2.movieTitle = S1.movieTitle)
```

Translate this query into a logical query plan, remove redundant joins from it, and then use heuristics to further optimize the obtained logical plan.

Optimization of logical query plans

An in-class integrated exercise

SQL

```
SELECT S1.movieTitle FROM StarsIn S1
WHERE S1.starName IN (SELECT name
                      FROM MovieStar, StarsIn S2
                      WHERE birthdate = 1960
                      AND S2.movieTitle = S1.movieTitle)
```

Translated logical query plan

$$\pi_{S1.\texttt{movieTitle}}\ \pi_{S1.*,\texttt{name}}\ \sigma_{S2.\texttt{movieTitle=S1.movieTitle} \wedge \texttt{birthdate=1960} \wedge S1.\texttt{starName}=\texttt{MovieStar.name}} (\texttt{Moviestar} \times \rho_{S2}(\texttt{StarsIn}) \times \rho_{S1}(\texttt{StarsIn}))$$

Optimization of logical query plans

An in-class integrated exercise

- MovieStar($\underline{\text{name: }\texttt{string}}$, address: `string`, gender: `char`, birthdate: `date`)
- StarsIn($\underline{\text{movieTitle: }\texttt{string}\text{, movieYear: }\texttt{string}\text{, starName: }\texttt{string}}$)

Translated logical query plan

$$\pi_{S1.\texttt{movieTitle}}\ \pi_{S1.*,\texttt{name}}\ \sigma_{S2.\texttt{movieTitle=S1.movieTitle}\wedge\texttt{birthdate=1960}\wedge S1.\texttt{starName}=\texttt{MovieStar.name}} (\texttt{Moviestar} \times \rho_{S2}(\texttt{StarsIn}) \times \rho_{S1}(\texttt{StarsIn}))$$

Conjunctive query:

$$Q(t) \leftarrow \texttt{MovieStar}(n, a, g, 1960), \texttt{StarsIn}(t, y_2, n_2), \texttt{StarsIn}(t, y, n)$$

Optimization of logical query plans

An in-class integrated exercise

Conjunctive query:

$$Q(t) \leftarrow \texttt{MovieStar}(n, a, g, 1960), \texttt{StarsIn}(t, y_2, n_2), \texttt{StarsIn}(t, y, n)$$

- the atom $\texttt{MovieStar}(n, a, g, 1960)$ cannot be removed (it is the only atom for relation `MovieStar`).
- the atom $\texttt{StarsIn}(t, y_2, n_2)$ can be removed since $Q \subseteq Q_1$ and $Q_1 \subseteq Q$ where:

$$Q_1(t) \leftarrow \texttt{MovieStar}(n, a, g, 1960), \texttt{StarsIn}(t, y, n)$$

Optimization of logical query plans

An in-class integrated exercise

We proceed with the conjunctive query:

$$Q_1(t) \leftarrow \mathtt{MovieStar}(n, a, g, 1960), \mathtt{StarsIn}(t, y, n)$$

- the only remaining atom $\mathtt{StarsIn}(t, y, n)$ cannot be removed (it is the only atom for relation StarsIn and it is the only atom containing the head variable t).

Optimization of logical query plans

An in-class integrated exercise

The minimal conjunctive query is hence:

$$Q_1(t) \leftarrow \texttt{MovieStar}(n, a, g, 1960), \texttt{StarsIn}(t, y, n)$$

Corresponding relational algebra expression:

$$\pi_{S.\texttt{movieTitle}} \sigma_{M.\texttt{birthdate=1960} \wedge S.\texttt{starName}=M.\texttt{name}} (\rho_M(\texttt{Moviestar}) \times \rho_S(\texttt{StarsIn}))$$

Optimization of logical query plans

An in-class integrated exercise

The minimal minimal relational algebra expression is hence:

$$\pi_{S.\texttt{movieTitle}}\sigma_{M.\texttt{birthdate=1960}\wedge S.\texttt{starName}=M.\texttt{name}} (\rho_M(\texttt{Moviestar}) \times \rho_S(\texttt{StarsIn}))$$

Recognize joins:

$$\pi_{S.\texttt{movieTitle}}\sigma_{M.\texttt{birthdate=1960}}(\rho_M(\texttt{Moviestar})) \bowtie_{S.\texttt{starName}=M.\texttt{name}} \rho_S(\texttt{StarsIn})$$

Push selections:

$$\pi_{S.\texttt{movieTitle}}(\sigma_{M.\texttt{birthdate=1960}}(\rho_M(\texttt{Moviestar})) \bowtie_{S.\texttt{starName}=M.\texttt{name}} \rho_S(\texttt{StarsIn}))$$

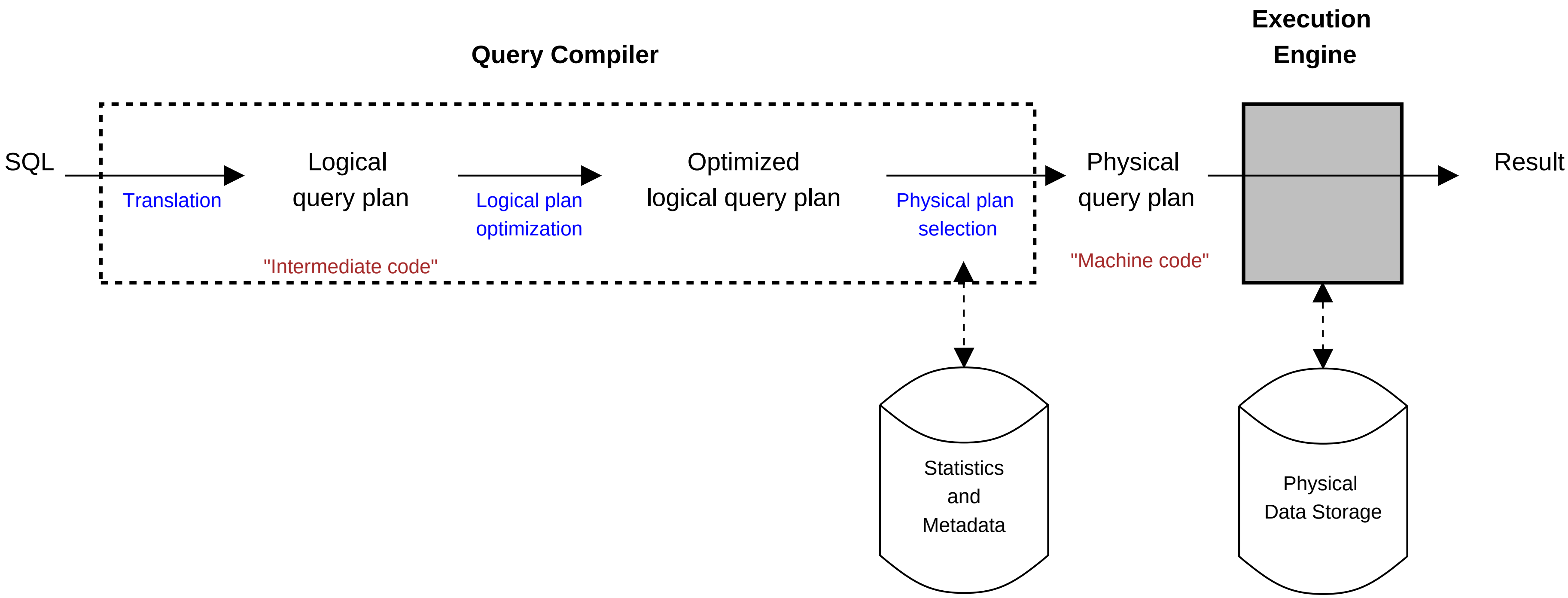
Add projections:

$$\pi_{S.\texttt{movieTitle}} (\sigma_{M.\texttt{birthdate=1960}}(\pi_{M.\texttt{birthdate},M.\texttt{name}}\rho_M(\texttt{Moviestar})) \bowtie_{S.\texttt{starName}=M.\texttt{name}} \pi_{S.\texttt{movieTitle},S.\texttt{starName}}\rho_S(\texttt{StarsIn}))$$

Physical data organization

Disks, blocks, tuples, schemas

Physical data organization



In order to select a physical plan we need to know:

- The physical algorithms available to implement the relational algebra operators

  e.g., scan a relation to implement a selection

- The situations in which each algorithm is best applied (situation x calls for algorithm A, situation y calls for algorithm B, $\ldots$).

Physical data organization

Physical algorithms depend on

- The representation of data on disk
- The data structures used

We hence need to know how data is physically organized before studying algorithms

This is the subject of chapters 13 and 14 in the book

Physical data organization

Database Management System

- Are responsible for enormous quantities of data (current scale: exabytes = 1 million gigabytes)
- Must query this data as efficiently as possible
- Must store data as reliably as possible

Hence we should wonder:

- What are the available storage media?
- How much "time" does it take to read from/write to these media?
- How can we minimize this costs?
- How can we prevent data loss due to disk crashes?

The answers to these questions may be found in chapter 13

Physical data organization

The types of data that we will need to store are:

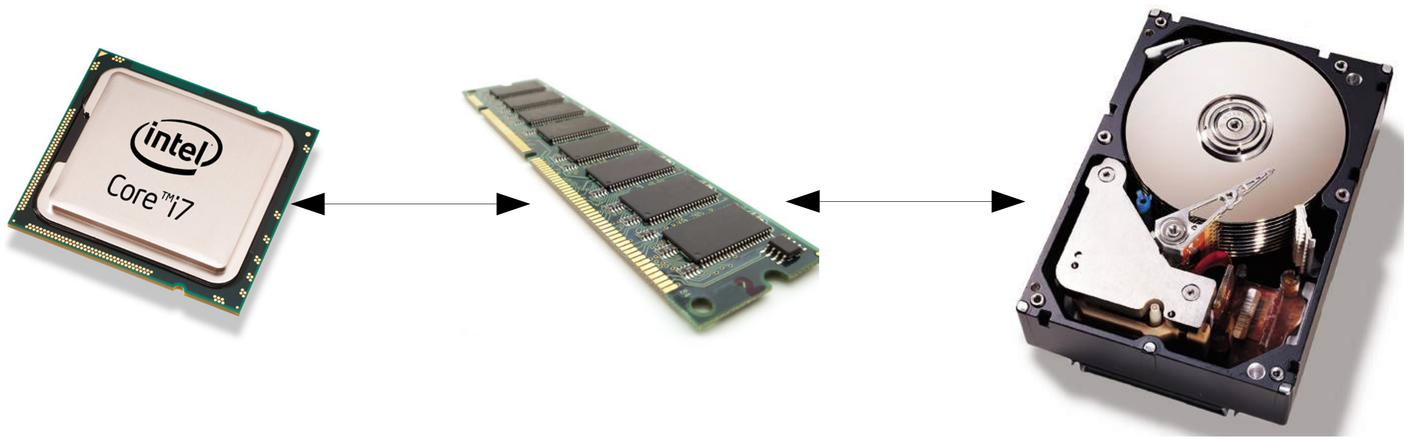
- Schemas
- Records
- Relations

How can we represent them efficiently "on disk"?

The answer may be found in chapter 13

One-dimensional index structures

Motivation: The I/O model of computation

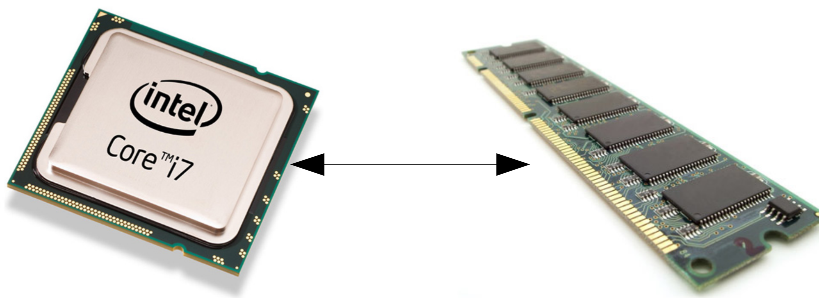


The I/O model

- Data is stored on disk, which is divided into blocks of bytes (typically 4 kilobytes)

  (each block can contain many data items)
- The CPU can only work on data items that are in memory, not on items on disk
- Therefore, data must first be transferred from disk to memory
- Data is transferred from disk to memory (and back) in whole blocks at the time
- The disk can hold D blocks, at most M blocks can be in memory at the same time (with $M << D$).

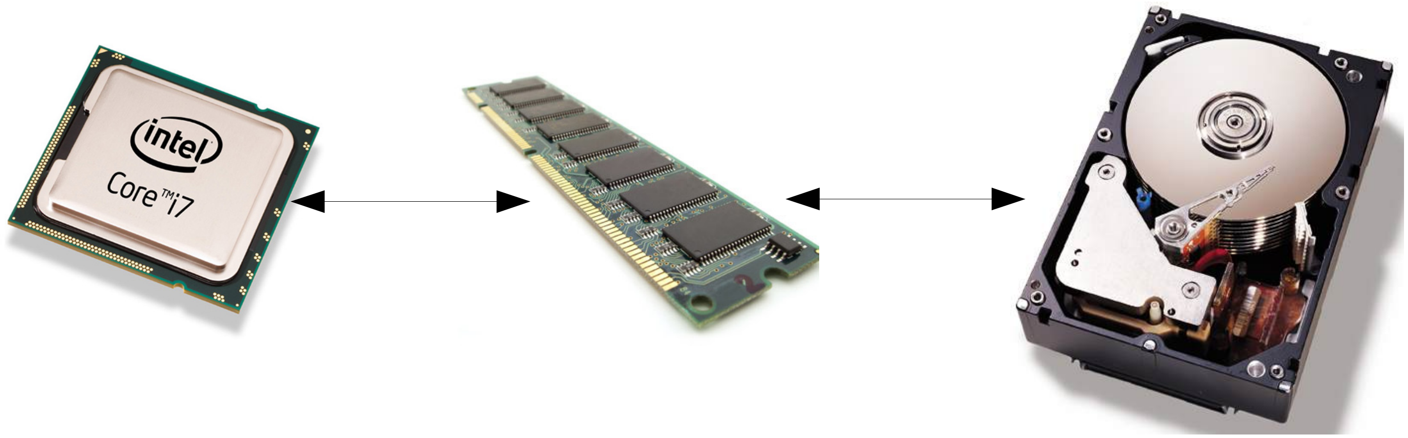
Motivation: The I/O model of computation



However: complexity of algorithms is traditionally analyzed in the RAM model of computation

- Data is stored in an (infinite) memory
- The CPU works on data items in memory
- Complexity is measured in terms of the number of memory accesses and CPU operations.

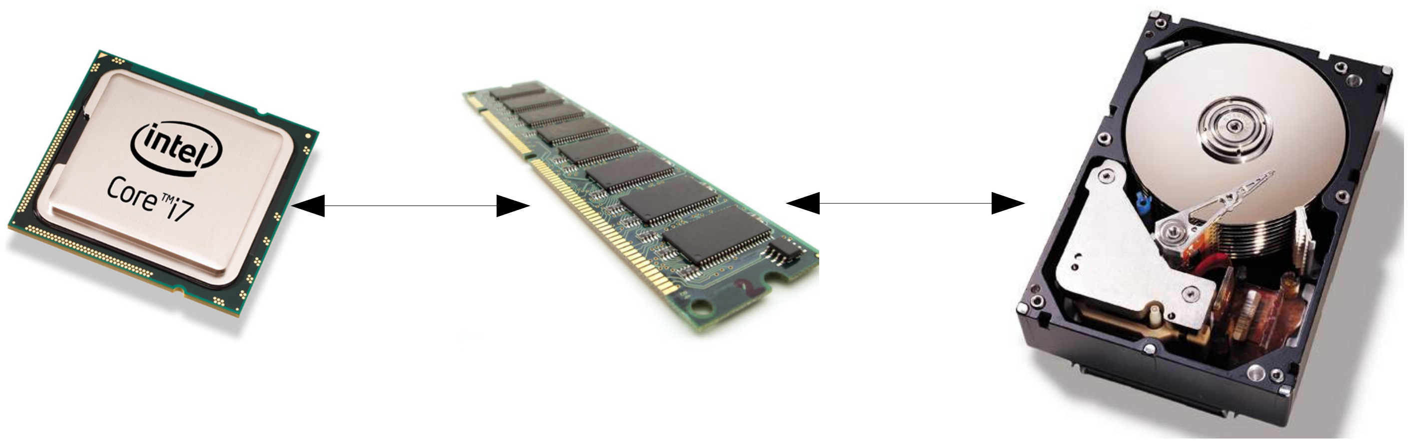
Motivation: The I/O model of computation



“The difference in speed between modern CPU and disk technologies is analogous to the difference in speed in sharpening a pencil using a sharpener on ones desk or by taking an airplane to the other side of the world and using a sharpener on someone elses desk.”

(D. Comer)

Motivation: The I/O model of computation



- In-memory computation is fast (memory access latency $\approx 10^{-8}s$)
- Disk-access is slow (HDD disk access latency: $\approx 10^{-3}s$, SSD: $\approx 10^{-5}s$)
- Hence: execution time is dominated by disk I/O

We will use the number of I/O operations required as cost metric

Intermezzo: Understanding memory and disk performance

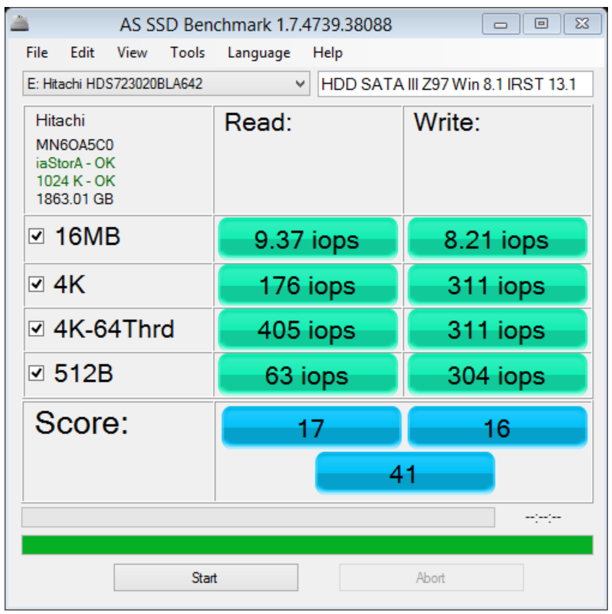
The performance of storage devices (including memory) is measured using three metrics:

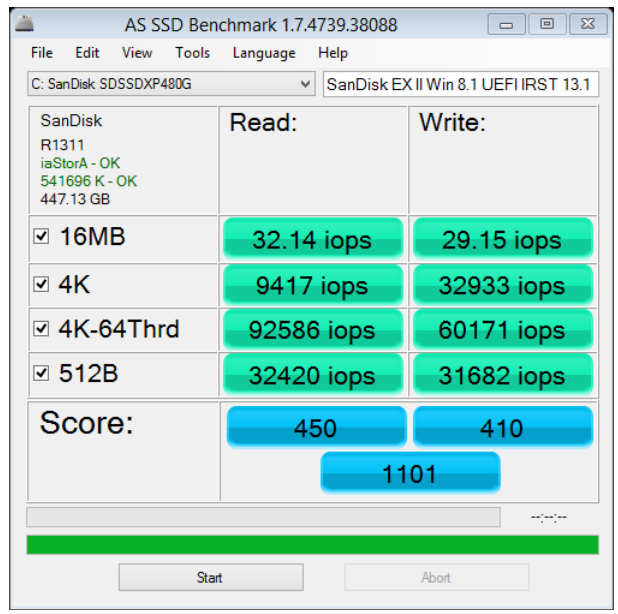
- Access latency: How long it takes for a storage device to start an I/O task (measured in seconds)
- Transfer rate (a.k.a. throughput or bandwidth): The speed at which data is transferred out of or into the storage device, once it has started (measured in MB/s)
- For a given block size, how often a storage device can perform I/O tasks of that block size is measured in Input/Output Operations per Second (IOPS).

Intermezzo: Understanding memory and disk performance

Some typical values:

	memory	HDD	SSD
Access latency	$\approx 10^{-8}$ s	$\approx 10^{-3}$ s	$\approx 10^{-5}$ s
Throughput	20 GB/s	100-200 MB/s	500-600 MB/s





Motivation: searching in a database

A hypothetical database

- A relation $R(A, B, C, D)$. Each tuple comprises 32 bytes.
- Attribute C is a (secondary) key for R.
- There are $128 \cdot 10^6$ tuples in the relation. The block size $B = 4096$ bytes.
- Hence there are 128 tuples per block, or 10^6 blocks in total.

Searching for record with $C = 10$ in case R is arbitrary

- For every block X in R:
  - Load X from disk in memory
  - Check whether there is a tuple with $A = 10$ in X;
  - If so output record and terminate loop; otherwise continue
  - Release X from memory
- Worst case I/O Cost: the total number of blocks in R, or 10^6 I/O's.
- At 10^{-3} s per IO this takes 16.6 minutes. $\Rightarrow$ **Can we do better?**

Index structures

See corresponding slides

Searching in a database with a index (1/2)

The database

- A relation $R(A, B, C, D)$. Each tuple comprises 32 bytes.
- Attribute C is a (secondary) key for R.
- There are $128 \cdot 10^6$ tuples in the relation. The block size $B = 4096$ bytes.
- Hence there are 128 tuples per block, or 10^6 blocks in total.

The index

- There is a secondary index on attribute C.
- A $(\texttt{key value}, \texttt{ptr})$ pair in the index takes 16 bytes.
- **Question:** How many $(\texttt{key}, \texttt{ptr})$ pairs fit in a block?
- **Question:** How many blocks does the dense 1st level index take?
- **Question:** How many blocks does the sparse 2nd level index take?

Searching in a database with a index (1/2)

The database

- A relation $R(A, B, C, D)$. Each tuple comprises 32 bytes.
- Attribute C is a (secondary) key for R.
- There are $128 \cdot 10^6$ tuples in the relation. The block size $B = 4096$ bytes.
- Hence there are 128 tuples per block, or 10^6 blocks in total.

The index

- There is a secondary index on attribute C.
- A $(\texttt{key value}, \texttt{ptr})$ pair in the index takes 16 bytes.
- **Question:** How many $(\texttt{key}, \texttt{ptr})$ pairs fit in a block? 256
- **Question:** How many blocks does the dense 1st level index take? $5 \cdot 10^5$
- **Question:** How many blocks does the sparse 2nd level index take? 1954

Searching in a database with a index (2/2)

Searching for records with $C = 10$ using the index

- Algorithm:
  - Loop through all of the blocks X in sparse index, one, by one, and find the $(\texttt{key}, \texttt{ptr})$ pair in X with the largest `key` value satisfying $\texttt{key} <= 10$.
  - Follow `ptr` to dense index block, and use the information in this block to locate the block in R containing the record with $C = 10$ (if it exists).
- Worst case I/O Cost: loading of all blocks of sparse index + 1 block of dense index + 1 block of R, or $1954 + 1 + 1 = 1956$ I/Os.
- At 10^{-3} s per I/O this takes 2 seconds.

Since the sparse index is sorted, we could perform binary search on it if it is sequential.

- I/O Cost: binary search in sparse index + 1 block of dense index + 1 block of R, or $\log_2(1954) + 1 + 1 = 14$ I/Os $\rightarrow 0.014$ seconds.

Searching in a database with a BTree index (1/6)

The database

- A relation $R(A, B, C, D)$. Attribute C is a (secondary) key for R.
- There are $128 \cdot 10^6$ tuples in the relation. The block size $B = 4096$ bytes.

The index

- There is a BTree index on attribute C.
- A `key value` takes 8 bytes, a `ptr` also 8 bytes.
- **Question:** What is the maximum order n of the BTree, taking into account that blocks are 4096 bytes large?

Searching in a database with a BTree index (2/6)

The database

- A relation $R(A, B, C, D)$. Attribute C is a (secondary) key for R.
- There are $128 \cdot 10^6$ tuples in the relation. The block size $B = 4096$ bytes.

The index

- There is a BTree index on attribute C.
- A `key value` takes 8 bytes, a `ptr` also 8 bytes.
- **Question:** What is the maximum order n of the BTree, taking into account that blocks are 4096 bytes large?
- **Answer:** A BTree of order n stores $n + 1$ pointers and n key values in each block. We are hence looking for the largest integer value of n satisfying:

$$(n + 1) \text{ ptrs } \times 8 \text{ bytes/ptr } + n \text{ keys } \times 8 \text{ bytes/ptr } \leq 4096 \text{ bytes}$$

As such, $n = 255$: we store 256 pointers and 255 keys in a block.

Searching in a database with a BTree index (3/6)

The database

- A relation $R(A, B, C, D)$. Attribute C is a (secondary) key for R.
- There are $128 \cdot 10^6$ tuples in the relation. The block size $B = 4096$ bytes.

The index

- There is a BTree index of order 255 on attribute C.
- **Question:** What is the height of the BTree assuming that leaf blocks are full and internal blocks contain 255 pointers?

Searching in a database with a BTree index (4/6)

The database

- A relation $R(A, B, C, D)$. Attribute C is a (secondary) key for R.
- There are $128 \cdot 10^6$ tuples in the relation. The block size $B = 4096$ bytes.

The index

- There is a BTree index of order 255 on attribute C.
- **Question:** What is the height of the BTree assuming that leaf blocks are full and internal blocks contain 255 pointers?
- **Answer:** : Observe:
  - there are $\left\lceil \frac{128 \cdot 10^6}{255} \right\rceil$ leaf blocks (at level 1)

Searching in a database with a BTree index (4/6)

The database

- A relation $R(A, B, C, D)$. Attribute C is a (secondary) key for R.
- There are $128 \cdot 10^6$ tuples in the relation. The block size $B = 4096$ bytes.

The index

- There is a BTree index of order 255 on attribute C.
- **Question:** What is the height of the BTree assuming that leaf blocks are full and internal blocks contain 255 pointers?
- **Answer:** : Observe:
  - there are $\left\lceil \frac{128 \cdot 10^6}{255} \right\rceil$ leaf blocks (at level 1)
  - there are $\left\lceil \frac{128 \cdot 10^6}{(255)^2} \right\rceil$ blocks at level 2

Searching in a database with a BTree index (4/6)

The database

- A relation $R(A, B, C, D)$. Attribute C is a (secondary) key for R.
- There are $128 \cdot 10^6$ tuples in the relation. The block size $B = 4096$ bytes.

The index

- There is a BTree index of order 255 on attribute C.
- **Question:** What is the height of the BTree assuming that leaf blocks are full and internal blocks contain 255 pointers?
- **Answer:** : Observe:
  - there are $\left\lceil \frac{128 \cdot 10^6}{255} \right\rceil$ leaf blocks (at level 1)
  - there are $\left\lceil \frac{128 \cdot 10^6}{(255)^2} \right\rceil$ blocks at level 2
  - there are $\left\lceil \frac{128 \cdot 10^6}{(255)^3} \right\rceil$ blocks at level 3

Searching in a database with a BTree index (4/6)

The database

- A relation $R(A, B, C, D)$. Attribute C is a (secondary) key for R.
- There are $128 \cdot 10^6$ tuples in the relation. The block size $B = 4096$ bytes.

The index

- There is a BTree index of order 255 on attribute C.
- **Question:** What is the height of the BTree assuming that leaf blocks are full and internal blocks contain 255 pointers?
- **Answer:** : Observe:
  - So, there are $\left\lceil \frac{128 \cdot 10^6}{(255)^h} \right\rceil$ blocks at level h

  Since the root is at the level where there is only one block, we are looking for the smallest value of h such that $\left\lceil \frac{128 \cdot 10^6}{(255)^h} \right\rceil = 1$.
  So, $h = \lceil \log_{255} 128 \cdot 10^6 \rceil = 4$.

Searching in a database with a BTree index (5/6)

The database

- A relation $R(A, B, C, D)$. Attribute C is a (secondary) key for R.
- There are $128 \cdot 10^6$ tuples in the relation. The block size $B = 4096$ bytes.

The index

- There is a BTree index of order 255 on attribute C.
- **Observe:** The height of the BTree is the smallest when all blocks are full. It is the largest when all blocks are only half full (when each block has its minimum size).
- **Question:** What is the height of the BTree assuming that all blocks are only half full?

Searching in a database with a BTree index (5/6)

The database

- A relation $R(A, B, C, D)$. Attribute C is a (secondary) key for R.
- There are $128 \cdot 10^6$ tuples in the relation. The block size $B = 4096$ bytes.

The index

- There is a BTree index of order 255 on attribute C.
- **Observe:** The height of the BTree is the smallest when all blocks are full. It is the largest when all blocks are only half full (when each block has its minimum size).
- **Question:** What is the height of the BTree assuming that all blocks are only half full? **Answer**: Same reasoning as before:
  $= \lceil \log_{\mathbf{128}} 128 \cdot 10^6 \rceil = 4$

Searching in a database with a BTree index (6/6)

The database

- A relation $R(A, B, C, D)$. Attribute C is a (secondary) key for R.
- There are $128 \cdot 10^6$ tuples in the relation. The block size $B = 4096$ bytes.

The index

- There is a BTree index of order 255 on attribute C.
- Hence we can store at most 256 pointers; and 255 key values in a block.
- **Question:** What is the cost of searching for the record with $C = 10$ using this BTree, assuming the worst-case scenario that each block in the BTree is half full?

Searching in a database with a BTree index (6/6)

The database

- A relation $R(A, B, C, D)$. Attribute C is a (secondary) key for R.
- There are $128 \cdot 10^6$ tuples in the relation. The block size $B = 4096$ bytes.

The index

- There is a BTree index of order 255 on attribute C.
- Hence we can store at most 256 pointers; and 255 key values in a block.
- **Question:** What is the cost of searching for the record with $C = 10$ using this BTree, assuming the worst-case scenario that each block in the BTree is half full?

  Answer: height of the Bree in which blocks are half full + 1 I/O to access main file
  $= \lceil \log_{\mathbf{128}} 128 \cdot 10^6 \rceil + 1 = 5 \rightarrow$ at $10^{-3}s$ per I/O this takes 0.005 seconds.

Inserting in a BTree index

The database

- A relation $R(A, B, C, D)$. Attribute C is a (secondary) key for R.
- There are $128 \cdot 10^6$ tuples in the relation.

The index

- There is a BTree index of order 255 on attribute C.
- Hence we can store at most 256 pointers; and 255 key values in a block.
- **Question:** What is the cost of inserting a new record in this BTree, assuming the record is already in the main file, and assuming the worst-case scenario where each block in the BTree is full?

Inserting in a BTree index

The database

- A relation $R(A, B, C, D)$. Attribute C is a (secondary) key for R.
- There are $128 \cdot 10^6$ tuples in the relation.

The index

- There is a BTree index on attribute C.
- Hence we can store at most 256 pointers; and 255 key values in a block.
- **Question:** What is the cost of inserting a new record in this BTree, assuming the record is already in the main file, and assuming the worst-case scenario where each block in the BTree is full?

  Answer: in this scenario, we will need to split an existing block at each level, and create a new root.

Inserting in a BTree index

The database

- A relation $R(A, B, C, D)$. Attribute C is a (secondary) key for R.
- There are $128 \cdot 10^6$ tuples in the relation.

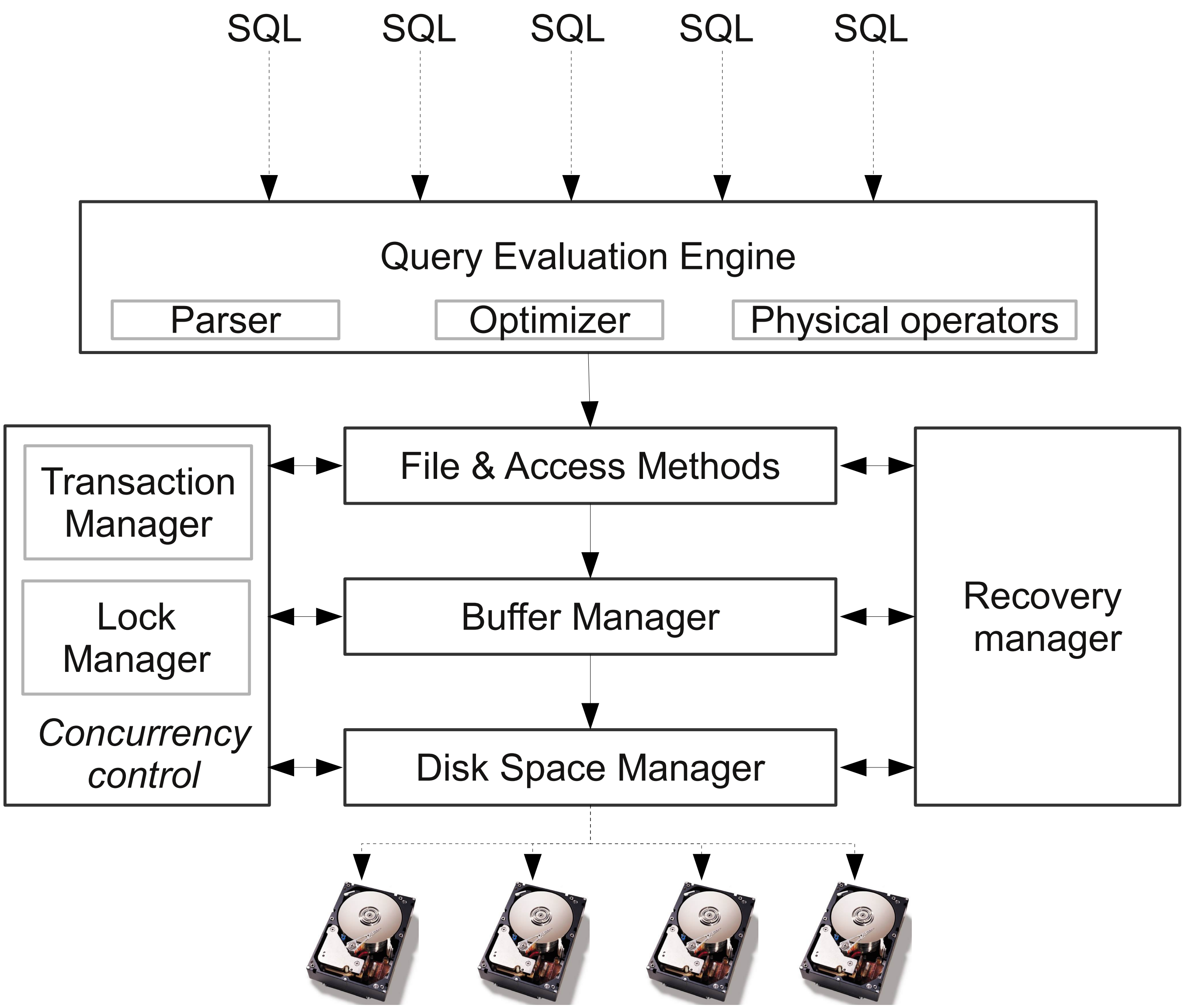
The index

- There is a BTree index on attribute C.
- Hence we can store at most 256 pointers; and 255 key values in a block.
- **Question:** What is the cost of inserting a new record in this BTree, assuming the record is already in the main file, and assuming the worst-case scenario where each block in the BTree is full?

  Answer: cost of a search + 2 I/O's per level of the BTree + new root
  $= \lceil \log_{255} 128 \cdot 10^6 \rceil + 2 \lceil \log_{255} 128 \cdot 10^6 \rceil + 1 = 3 \lceil \log_{255} 128 \cdot 10^6 \rceil + 1 = 13 \rightarrow 0.013s$

Intermezzo: A typical database architecture

A typical database architecture



A typical database architecture

Main components

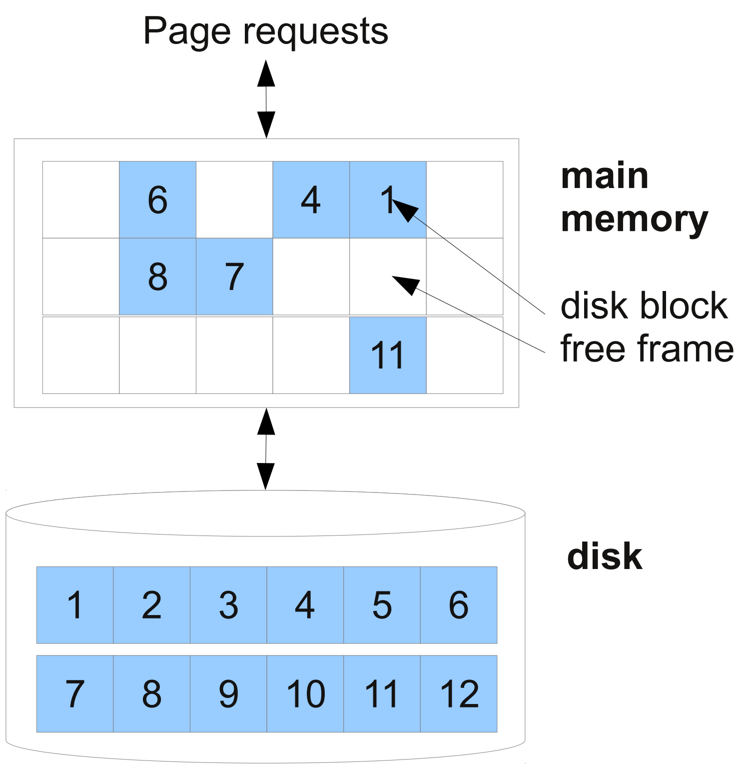
- The lowest layer of the DBMS software deals with management of space on disk, where the data is stored. Higher layers allocate, deallocate, read, and write blocks through (routines provided by) this layer, called the disk space manager or storage manager.
- The buffer manager brings blocks in from disk to main memory in response to read requests from the higher-level layers.
- The file and access methods layer supports the concept of reading and writing files (as collection of blocks or a collection of records) as well as indexes. In addition to keeping track of the blocks in a file, this layer is responsible for organizing the information within a block.
- The query evaluation engine, and more in particular the code that implements relational operators, sits on top of the file and access methods layer.
- The DBMS supports concurrency and crash recovery by carefully scheduling user requests and maintaining a log of all changes to the database. These tasks are managed by the concurrency control manager and the recovery manager.

A typical database architecture

Disk Space Manager

- The disk space manager manages space on disk.
- Abstractly, it supports the concept of a block as a unit of data and provides commands to allocate or deallocate a block and read or write a block.
- A database grows and shrinks when records are inserted and deleted over time. The disk space manager keeps track of which disk blocks are in use. Although it is likely that blocks are initially allocated sequentially on disk, subsequent allocations and deallocations could in general create 'holes.' One way to keep track of block usage is to maintain a list of free blocks. When blocks are deallocated, they are added to the free list for future use.
- The disk space manager hides details of the underlying hardware and operating system and allows higher levels of the software to think of the data as a collection of blocks.
- Although it typically uses the file system functionality provided by the OS, it provides additional features, like the possibility to distribute data on multiple disks, etc.

A typical database architecture



Buffer Manager

- Mediates between external storage and main memory
- Maintains a designated main memory area, called the buffer pool for this task.
- The buffer pool is a collection of memory slots where each slot (called a frame or buffer) can contain exactly one block.
- Disk blocks are brought into memory as needed in response to higher-level requests.
- A replacement policy decides which block to evict when the buffer is full.

A typical database architecture

Buffer Manager (continued)

- Higher levels of the DBMS code can be written without worrying about whether data blocks are in memory or not: they ask the buffer manager for the block, and the buffer manager loads it into a slot in the buffer pool if it is not already there.
- The higher-level code must also inform the buffer manager when it no longer needs a block that it has requested to be brought into memory. That way, the buffer manager can re-use the slot for future requests.
- A buffer whose block contents should remain in memory (e.g., because a routine from a higher-level layer is working with its contents) is called pinned. The act of asking the buffer manager to read a disk block into a buffer slot is called pinning and the act of letting the buffer manager know that a block is no longer needed in memory is called unpinning.
- When higher-level code unpins a block, it must also inform the buffer manager whether it modified the requested block; the buffer manager then makes sure that the change is eventually propagated to the copy of the block on disk.

A typical database architecture

Buffer Manager (continued)

- When the buffer manager receives a block pin request, it checks whether the block is already in memory (because another DBMS component is working on it, or because it was recently loaded but then unpinned). If so, the corresponding buffer is re-used and no disk I/O takes place.
- If not, the buffer manager has to decide a buffer frame to load the block into from disk. If there are no empty frames available, the buffer manager has to select a frame containing a block that is currently unpinned, write the contents of that block back to disk if modifications are made, and load the requested block from disk into the frame.
- The strategy by which the buffer manager chooses the slot to release back to disk is called the buffer replacement policy. Popular policies are FIFO, Least recently used, Clock.

A typical database architecture

Buffer Management in Reality

- Prefetching
  - Buffer managers try to anticipate page requests to overlap CPU and I/O operations.

    Speculative prefetching Assume sequential scan and automatically read ahead.

    Prefetch lists Some database algorithms can inform the buffer manager of a list of blocks to prefetch.
- Page fixing/hating
  - Higher-level code may request to fix a page if it may be useful in the near future (e.g., index pages).
  - Likewise, an operator that hates a page won't access it any time soon (e.g., table pages in a sequential scan).
- Multiple buffer pools
  - E.g., separate pools for indexes and tables.

Multi-dimensional index structures
Part I: motivation

Motivation: Data Warehouse

A definition

> *"A data warehouse is a repository of integrated enterprise data. A data warehouse is used specifically for decision support, i.e., there is (typically, or ideally) only one data warehouse in an enterprise. A data warehouse typically contains data collected from a large number of sources within, and sometimes also outside, the enterprise."*

Decision support (1/2)

'Traditional" relational databases were designed for online transaction processing (OLTP):

- flight reservations; bank terminal; student administration; ...

OLTP characteristics:

- Operational setting (e.g., ticket sales)
- Up-to-date = critical (e.g., do not book the same seat twice)
- Simple data (e.g., [reservation, date, name])
- Simple queries that only access a small part of the database (e.g., "Give the flight details of X" or "List flights to Y")

Decision support systems have different requirements.

Decision support (2/2)

Decision support systems have different requirements:

- Offline setting (e.g., evaluate flight sales)
- Historical data (e.g., flights of last year)
- Summarized data (e.g., # passengers per carrier for destination X)
- Integrates different databases (e.g., passengers, fuel costs, maintenance information)
- Complex statistical queries (e.g., average percentage of seats sold per month and destination)

Decision support (2/2)

Decision support systems have different requirements:

- Offline setting (e.g., evaluate flight sales)
- Historical data (e.g., flights of last year)
- Summarized data (e.g., # passengers per carrier for destination X)
- Integrates different databases (e.g., passengers, fuel costs, maintenance information)
- Complex statistical queries (e.g., average percentage of seats sold per month and destination)

Taking these criteria into mind, data warehouses are tuned for online analytical processing (OLAP)

- Online = answers are immediately available, without delay.

The Data Cube: Generalizing Cross-Tabulations

Cross-tabulations are highly useful for analysis

- Example: sales June to August 2010

	Blue	Red	Orange	Total
June	51	25	128	**234**
July	58	20	120	**198**
August	65	22	51	**138**
Total	**174**	**67**	**329**	**570**

The Data Cube: Generalizing Cross-Tabulations

Cross-tabulations are highly useful for analysis

Data Cubes are extensions of cross-tabs to multiple dimensions

Dimension X

Dimension Y

	Blue	Red	Orange	Total
June	51	25	128	**234**
July	58	20	120	**198**
August	65	22	51	**138**
Total	**174**	**67**	**329**	**570**

Aggregated w.r.t Dimension X

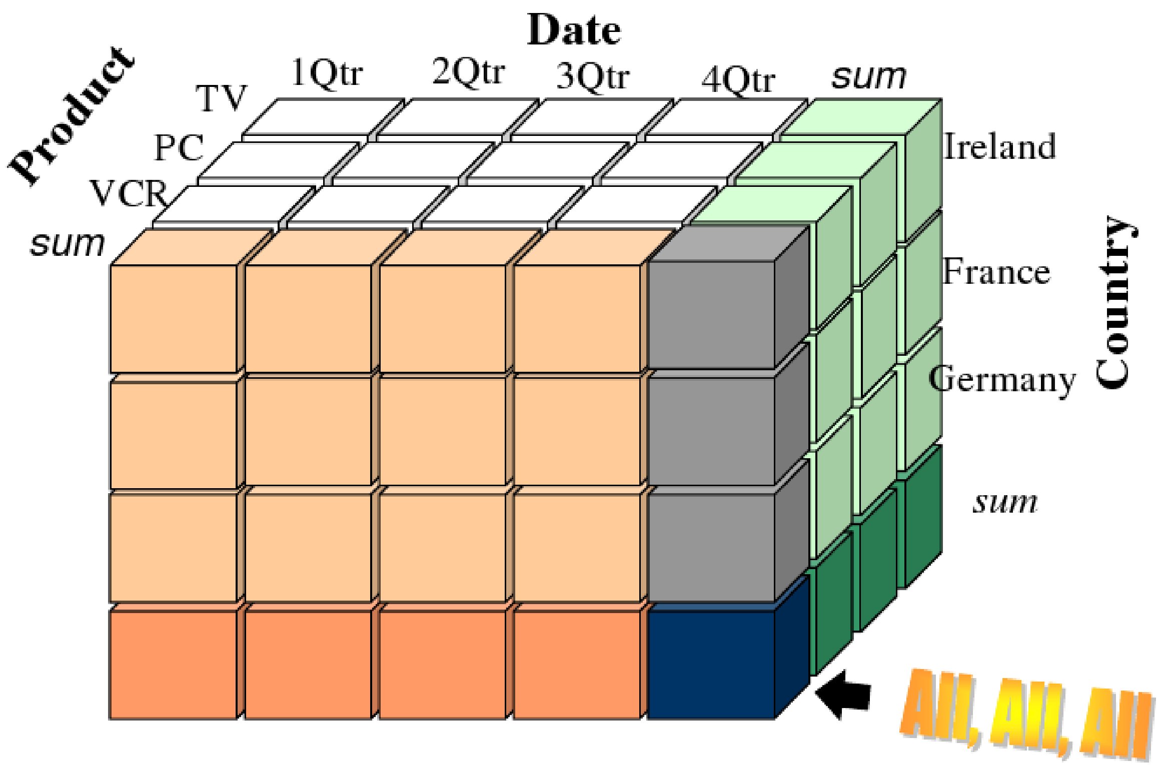
Aggregated w.r.t. Dimension Y

Aggregated w.r.t Dimension X and Y

The Data Cube: Generalizing Cross-Tabulations

Cross-tabulations are highly useful for analysis

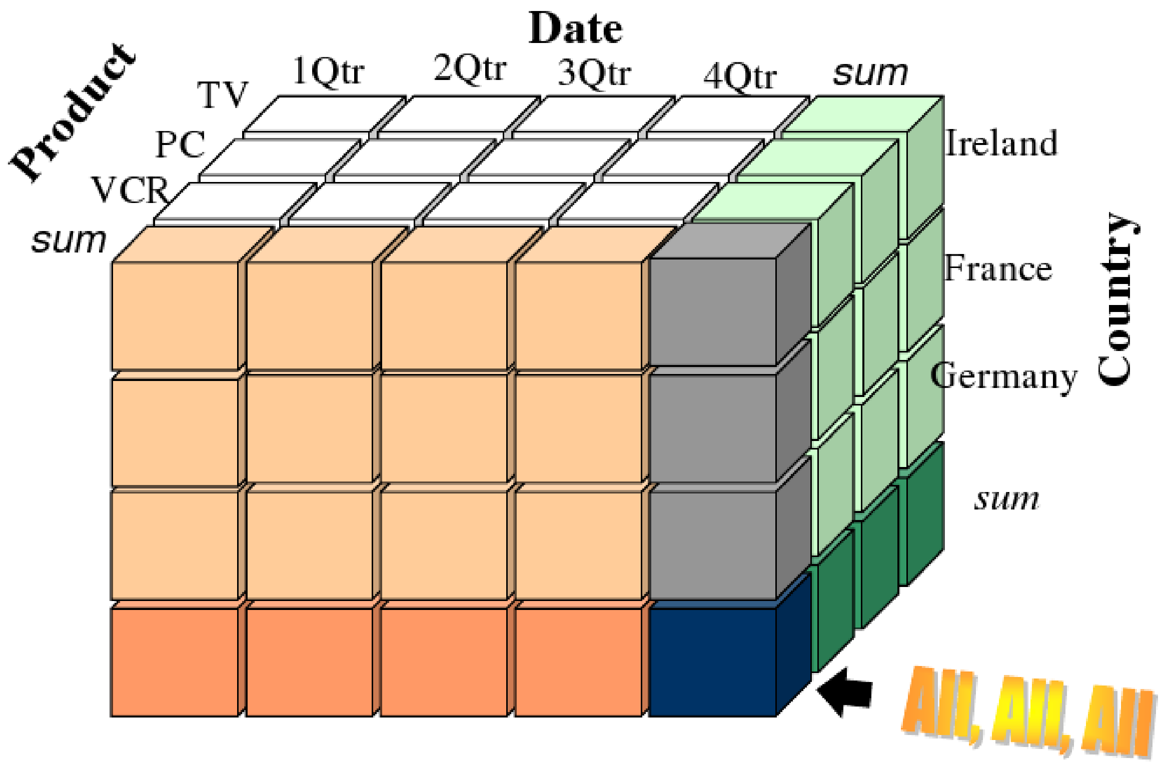
Data Cubes are extensions of cross-tabs to multiple dimensions



OLAP Operations on the CUBE

Roll-up

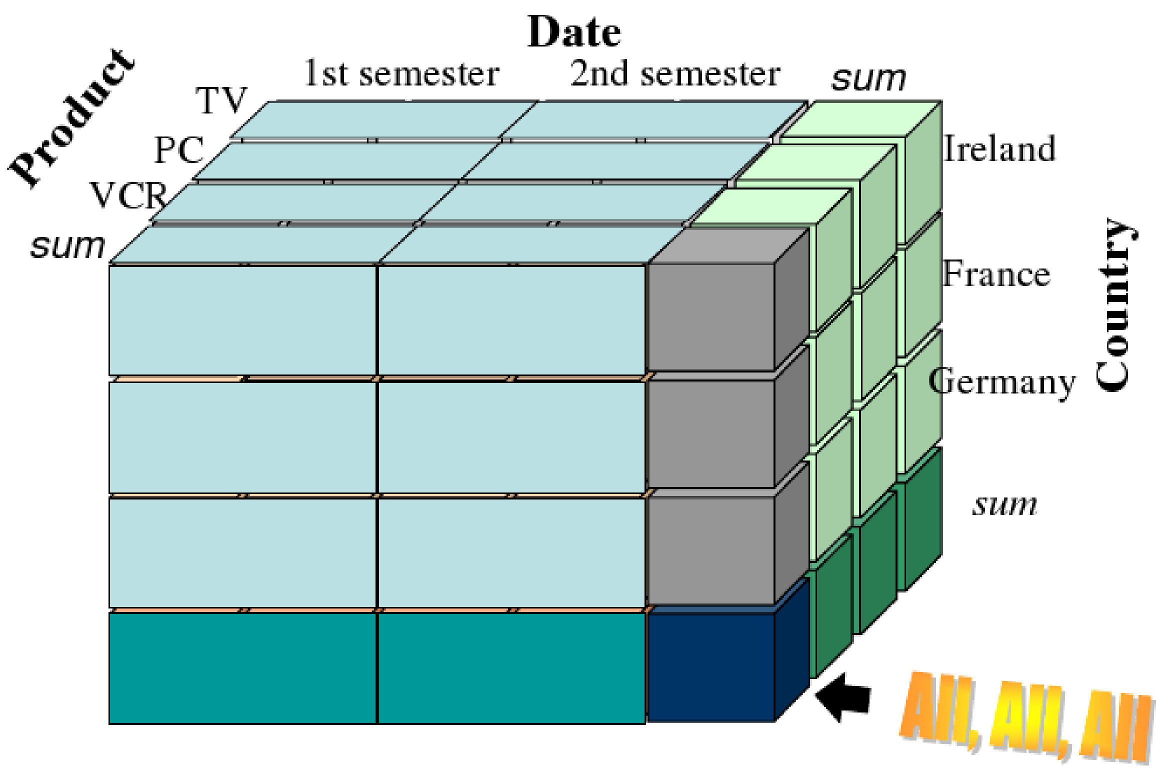
- Group per semester instead of per quarter



OLAP Operations on the CUBE

Roll-up

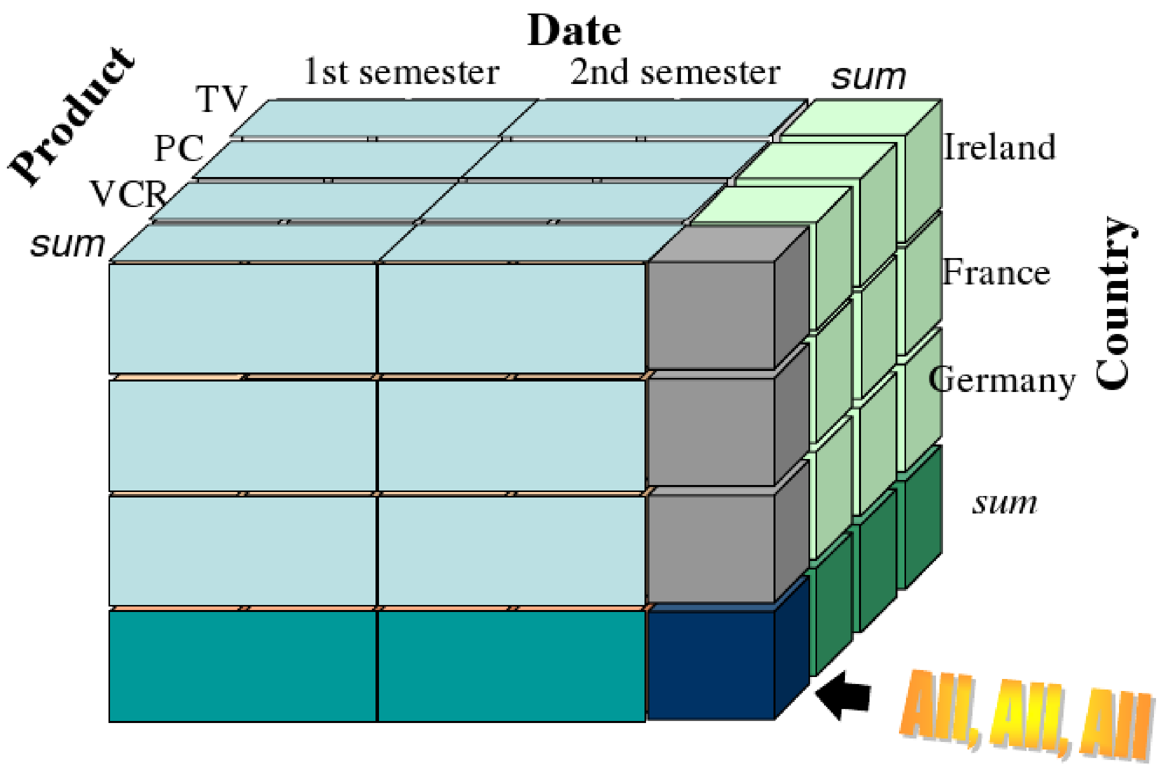
- Show me totals per semester instead of per quarter



OLAP Operations on the CUBE

Roll-up

- Show me totals per semester instead of per quarter

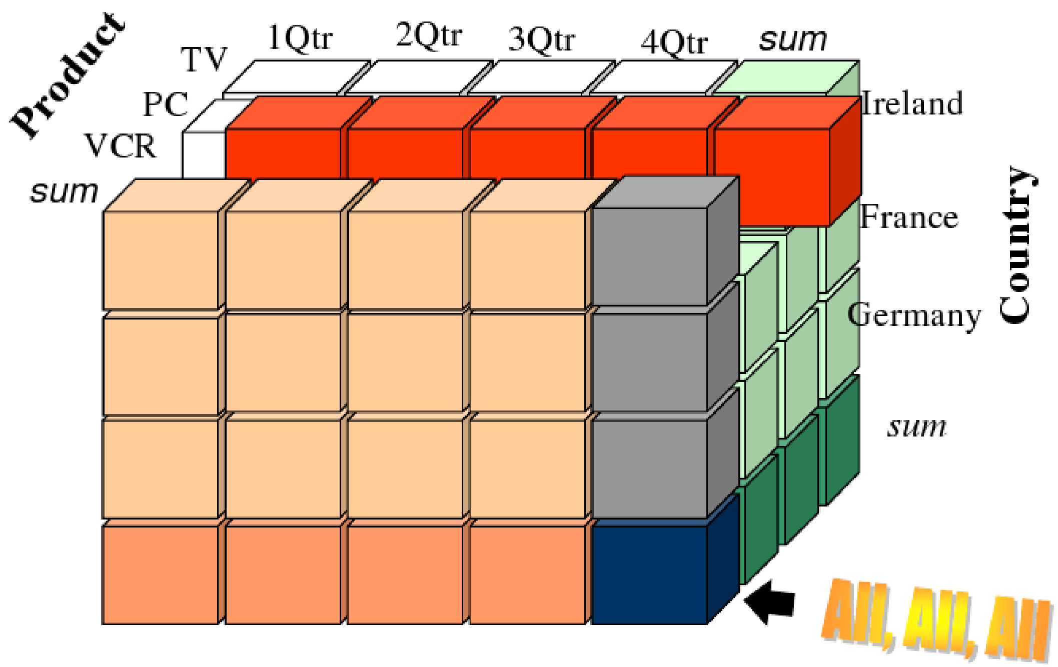


Inverse is drill-down

OLAP Operations on the CUBE

Slice and dice

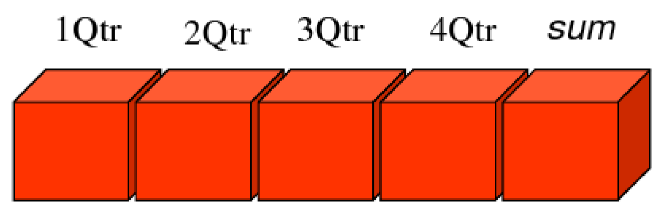
- Select part of the cube by restricting one or more dimensions
- E.g, restrict analysis to Ireland and VCR



OLAP Operations on the CUBE

Slice and dice

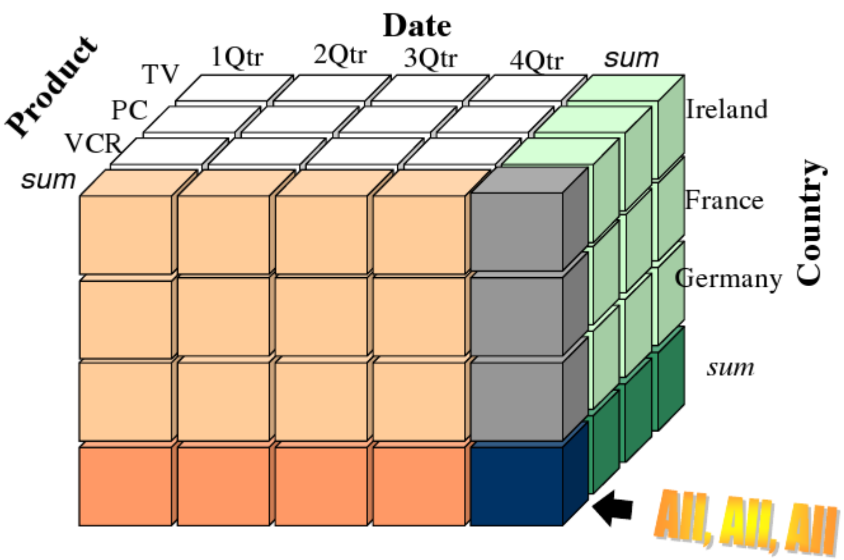
- Select part of the cube by restricting one or more dimensions
- E.g, restrict analysis to Ireland and VCR



Different OLAP systems

Multidimensional OLAP (MOLAP)

- Early implementations used a multidimensional array to store the cube completely:
- In particular: pre-compute and materialize all aggregations



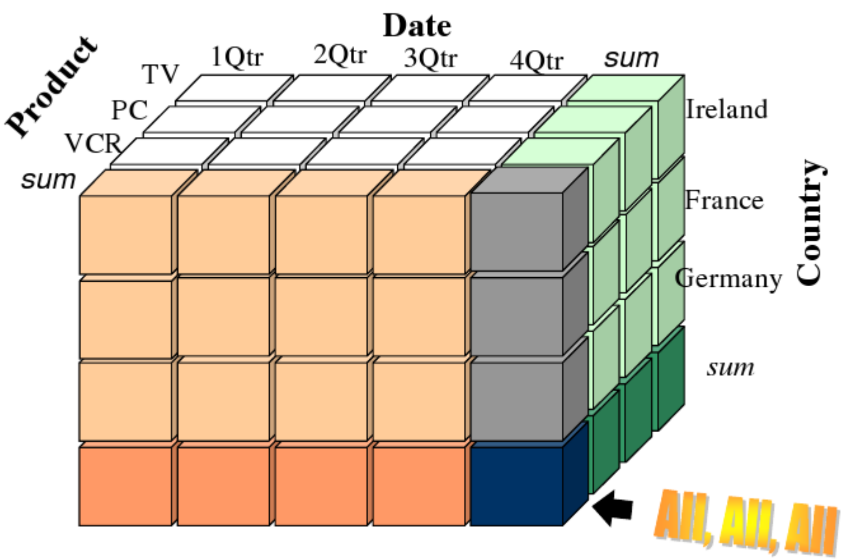
Array: **cell[product, date, country]**

- Fast lookup: to access cell[p,d,c] just use array indexation

Different OLAP systems

Multidimensional OLAP (MOLAP)

- Early implementations used a multidimensional array to store the cube completely:
- In particular: pre-compute and materialize all aggregations



Array: **cell[product, date, country]**

- Fast lookup: to access cell[p,d,c] just use array indexation
- Very quickly people realized that this is infeasible due to the data explosion problem

The data explosion problem

The problem:

- Data is not dense but sparse
- Hence, if we have n dimensions with each c possible values, then we do not actually have data for all the c^n cells in the cube.
- Nevertheless, the multidimensional array representation realizes space for all of these cells

The data explosion problem

The problem:

- Data is not dense but sparse
- Hence, if we have n dimensions with each c possible values, then we do not actually have data for all the c^n cells in the cube.
- Nevertheless, the multidimensional array representation realizes space for all of these cells

Example: 10 dimensions with 10 possible values each

- 10 000 000 000 cells in the cube
- suppose each cell is a 64-bit integer
- then the multidimensional-array representing the cube requires ≈ 74.5 gigabytes to store $\rightarrow$ does not fit in memory!
- yet if only 1 000 000 cells are present in the data, we actually only need to store ≈ 0.0074 gigabytes

Multidimensional OLAP (MOLAP)

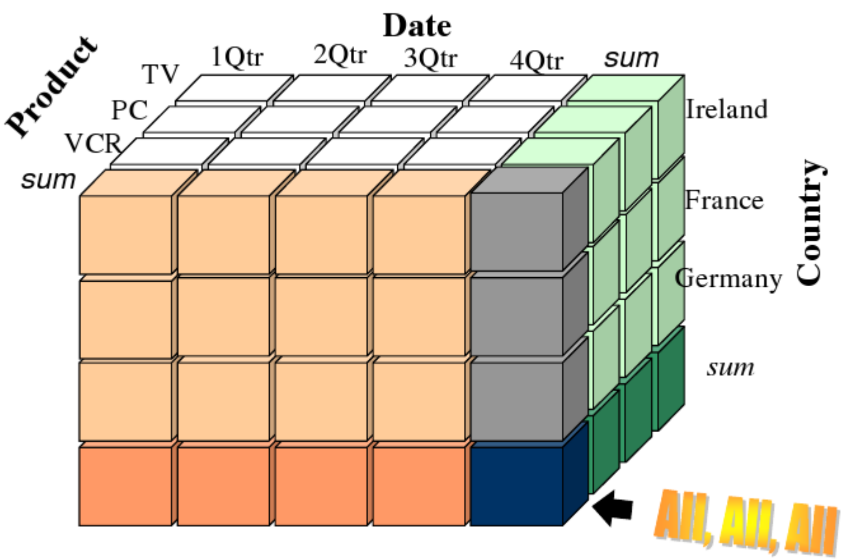
In conclusion

- Naively storing the entire cube does not work.
- Alternative representation strategies use sparse main memory index structures:
  - search trees
  - hash tables
  - ...
- And these can be specialized to also work in secondary memory
  $\rightarrow$ multidimensional indexes (the main technical content of this lecture).

Relational OLAP (ROLAP)

Key Insight [Gray et al, Data Mining and Knowledge Discovery, 1997]

- The n-dimensional cube can be represented as a traditional relation with $n + 1$ columns (1 column for each dimension, 1 column for the aggregate)
- Use special symbol `ALL` to represent grouping

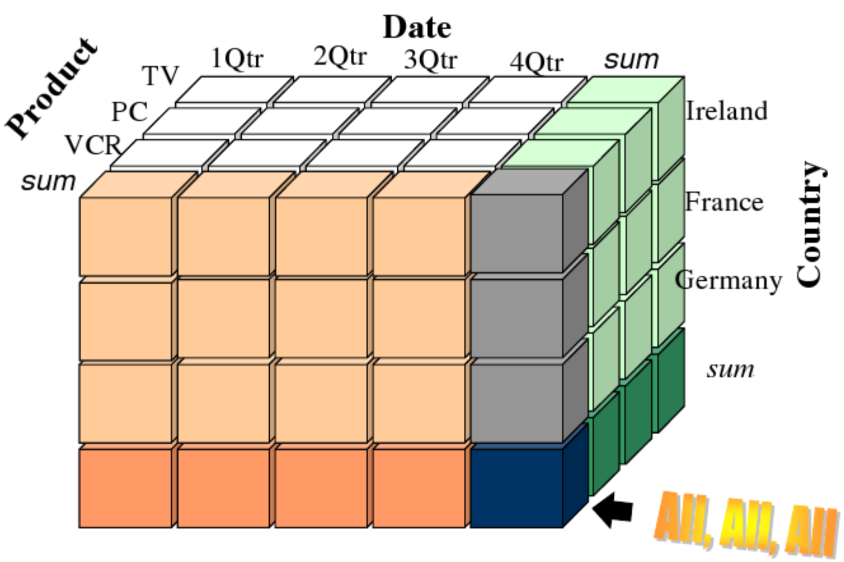


Product	Date	Country	Sales
TV	Q1	Ireland	100
TV	Q2	Ireland	80
TV	Q3	Ireland	35
...	...	...	...
PC	Q1	Ireland	100
...	...	...	...
TV	**ALL**	Ireland	215
TV	**ALL**	**ALL**	1459
...	...	...	...
ALL	**ALL**	**ALL**	109290

Relational OLAP (ROLAP)

Key benefits: space usage

- The non-aggregate cells that are not present in the original data are also not present in the relational cube representation.
- Moreover, it is straightforward to represent only aggregation tuples in which all dimension columns have values that already occur in the data



Product	Date	Country	Sales
TV	Q1	Ireland	100
TV	Q2	Ireland	80
TV	Q3	Ireland	35
...	...	...	...
PC	Q1	Ireland	100
...	...	...	...
TV	**ALL**	Ireland	215
TV	**ALL**	**ALL**	1459
...	...	...	...
ALL	**ALL**	**ALL**	109290

Relational OLAP (ROLAP)

Key benefits

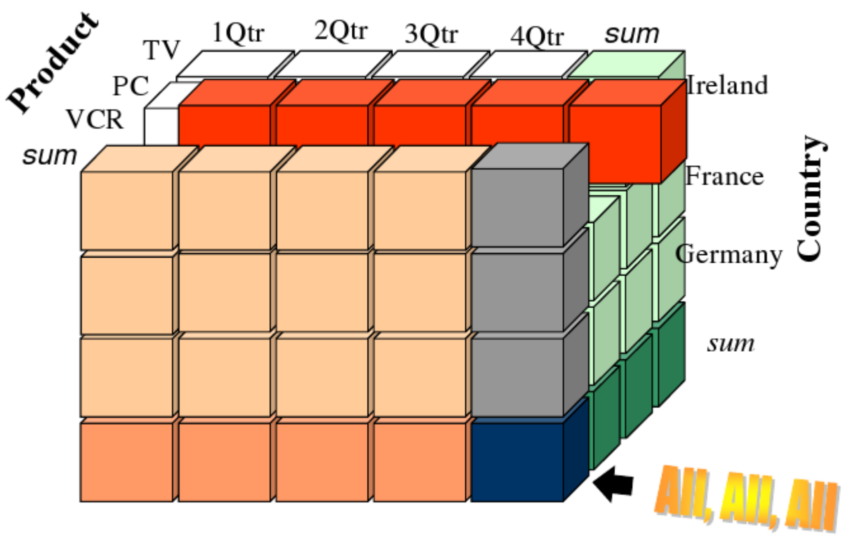
- By representing the cube as a relation it can be stored in a "traditional" relational DBMS ...
- ... which works in secondary memory by design (good for multi-terraby data warehouses) ...
- Hence one can re-use the rich literature on relational query storage and query evaluation techniques,

But, to be honest, much research was done to get this representation efficient in practice.

Relational OLAP (ROLAP)

Key benefits: use SQL

- Dice example: restrict analysis to Ireland and VCR



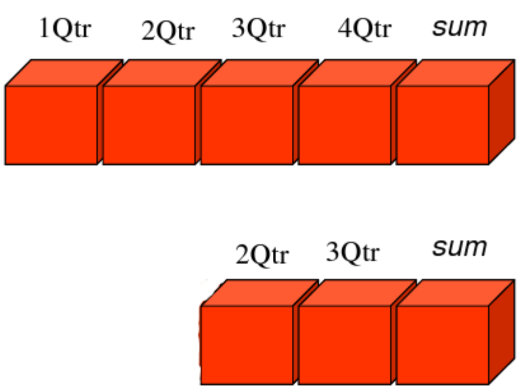

```
SELECT Date, Sales
FROM Cube_table
WHERE Product = "VCR"
  AND Country = "Ireland"
```

Date	Sales
Q1	100
Q2	80
Q3	35
ALL	215

Relational OLAP (ROLAP)

Key benefits: use SQL

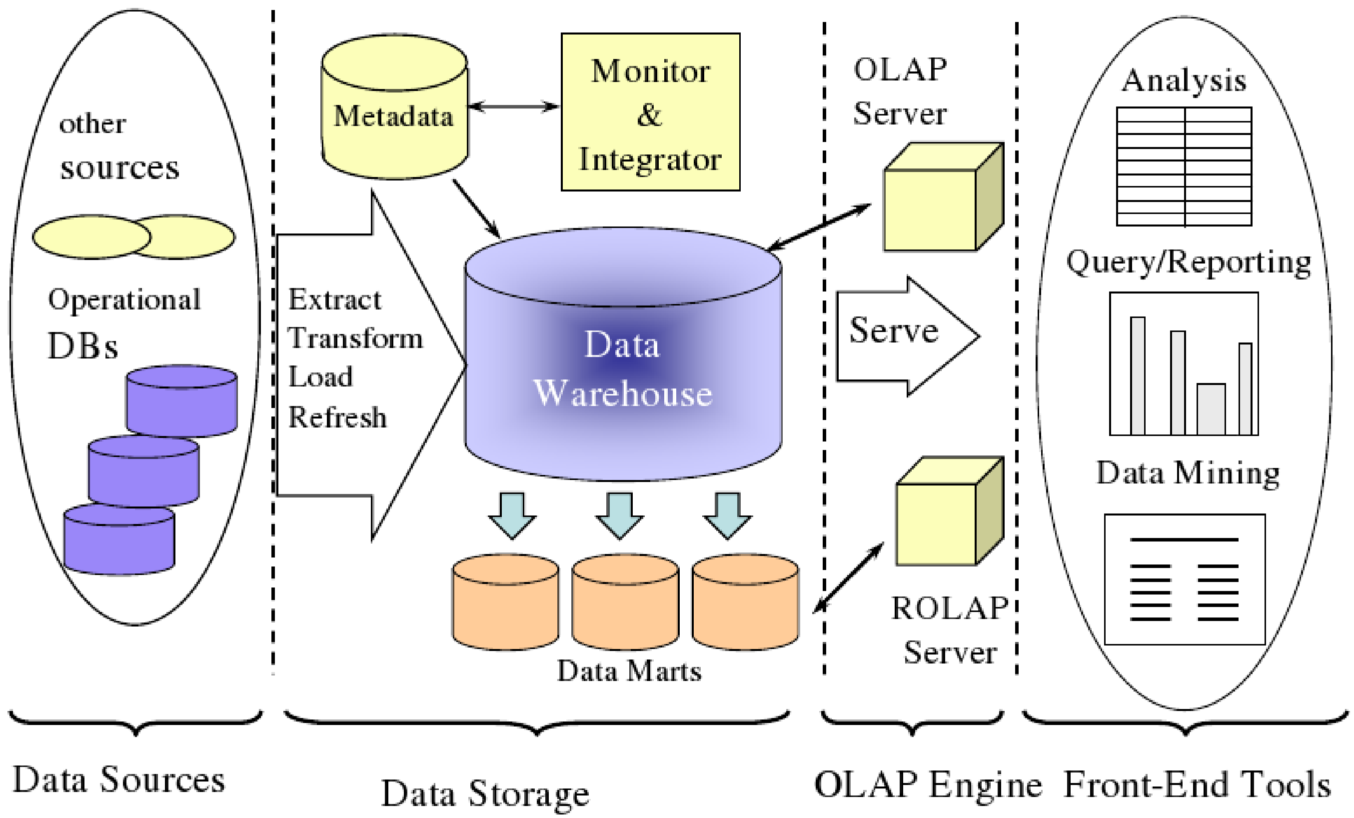
- Dice example: restrict analysis to Ireland and VCR, quarter 2 and quarter 3
  → need to compute a new total aggregate for this sub-cube




```
(SELECT Date, Sales
 FROM Cube_table
 WHERE Product = "VCR"
  AND Country = "Ireland"
  AND (Date = "Q2" OR Date = "Q3")
  AND SALES <> "ALL")
UNION
(SELECT "ALL" as DATE, SUM(T.Sales) as SALES
 FROM Cube_table t
 WHERE Product = "VCR"
  AND Country = "Ireland"
  AND (Date = "Q2" OR Date = "Q3")
  AND SALES <> "ALL"
 GROUP BY Product, Country)
```

This actually motivated the extension of SQL with CUBE-specific operators and keywords

Three-tier architecture



Multi-dimensional index structures
Part II: index structures

Multidimensional Indexes

Typical example of an application requiring multidimensional search keys:

Searching in the data cube and searching in a spatial database

Typical queries with multidimensional search keys:

- Point queries:
  - retrieve the Sales total for the product TV sold in Ireland, with an `ALL` value for date.
  - does there exist a star on coordinate $(10, 3, 5)$?
- Partial match queries: return the coordinates of all stars with $x = 5$ and $z = 3$.
- Dicing / Range queries:
  - return all cube cells with date $\geq$ Q1 and date $\leq$ Q3 and sales ≤ 100;
  - return the coordinates of all stars with $x >= 10$ and $20 \leq y \leq 35$.
- Nearest-neighbour queries: return the three stars closest to the star at coordinate $(10, 15, 20)$.

Multidimensional Indexes

Indexes for search keys comprising multiple attributes?

- BTree: assumes that the search keys can be ordered. What order can we put on multidimensional search keys?

  $\rightarrow$ Pick the lexicographical order:

$$\begin{aligned}(x, y, z) \leq (x', y', z') \Leftrightarrow\ & x < x' \\ & \vee (x = x' \wedge y < y') \\ & \vee (x = x' \wedge y = y' \wedge z \leq z')\end{aligned}$$

- Hash table: assumes a hash function h : *keys* $\rightarrow \mathbb{N}$. What hash function can we put on multidimensional search keys?

  $\rightarrow$ Extend the hash function to tuples:

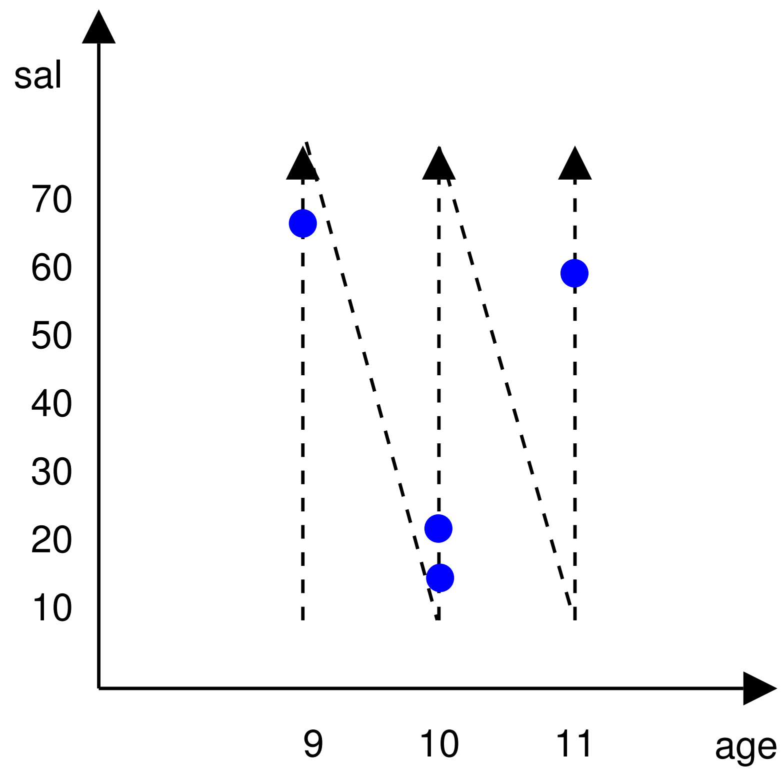
$$h(x, y, z) = h(x) + h(y) + h(z)$$

Multidimensional Indexes

Problem with the lexicographical order in BTrees:

Assume that we have a BTree index on (age, sal) pairs.

- age < 20: ok
- sal < 30: linear scan
- age $< 20 \wedge$ sal < 20



Multidimensional Indexes

Problem with hash tables:

Assume that we have a hash table on (age, sal) pairs.

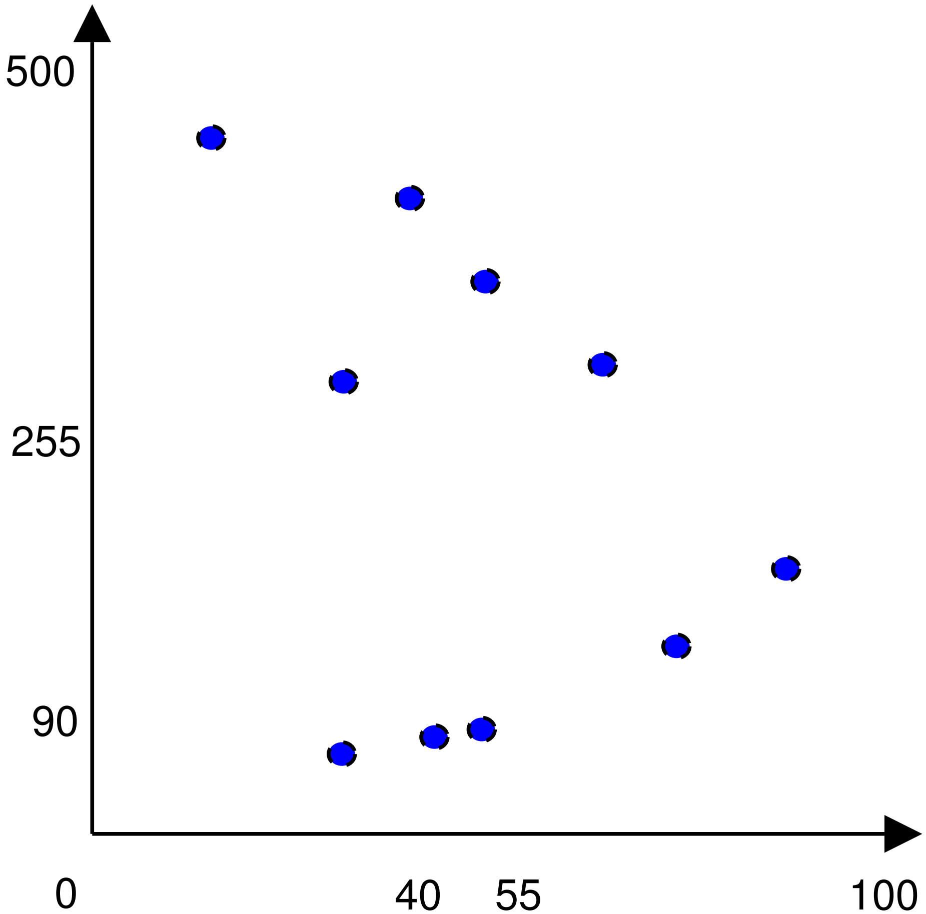
- age < 20: linear scan
- sal < 30: linear scan
- age $< 20 \wedge$ sal < 20: linear scan

Conclusion: for queries with multidimensional search keys we want to index points by spatial proximity

.

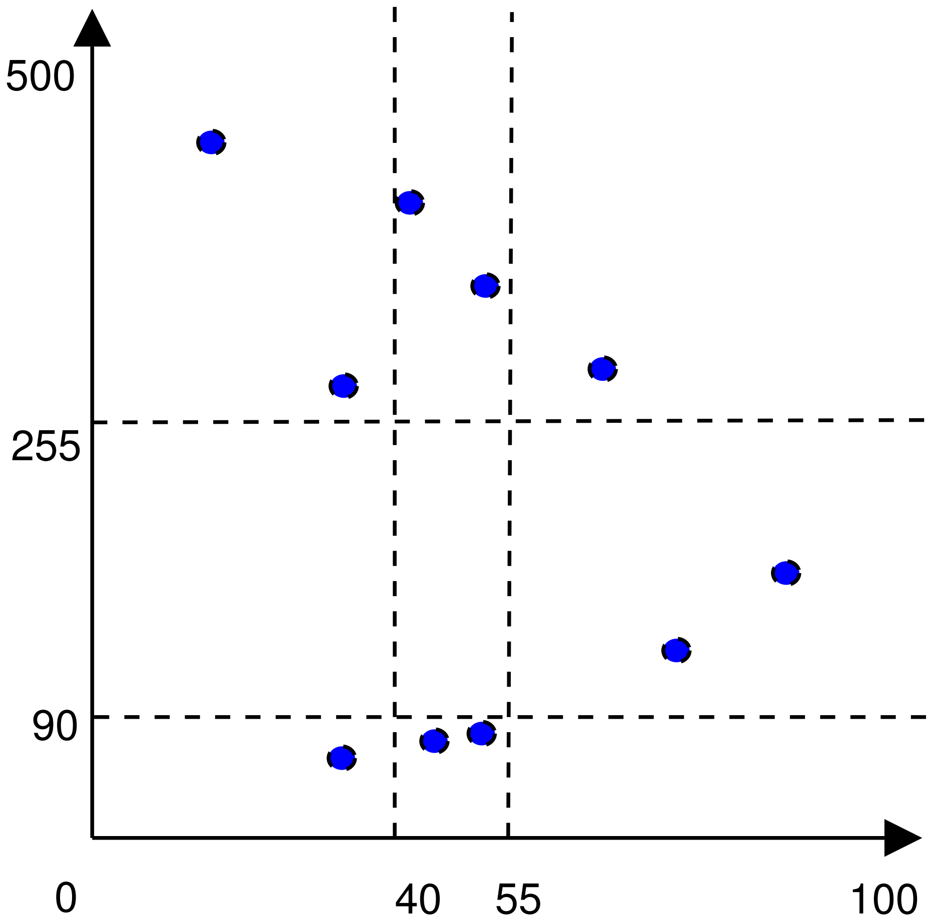
Multidimensional Indexes

Grid files: a variant on hashing



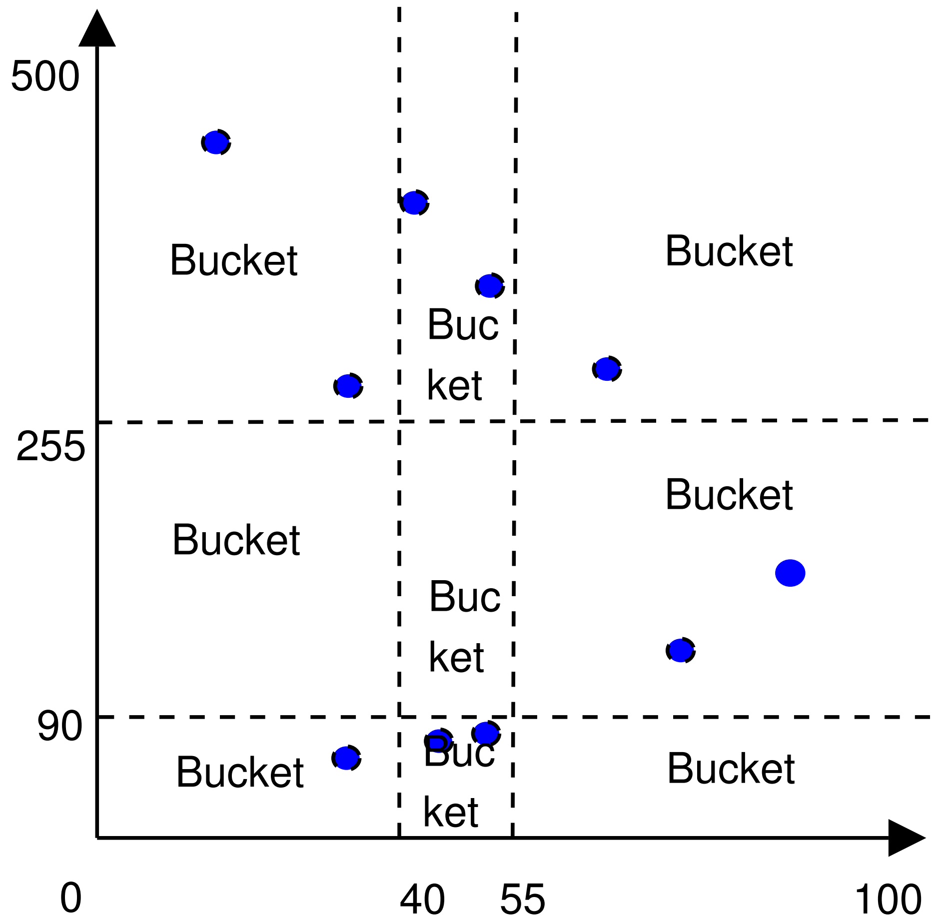
Multidimensional Indexes

Grid files: a variant on hashing



Multidimensional Indexes

Grid files: a variant on hashing



- Insert: find the corresponding bucket, and insert.

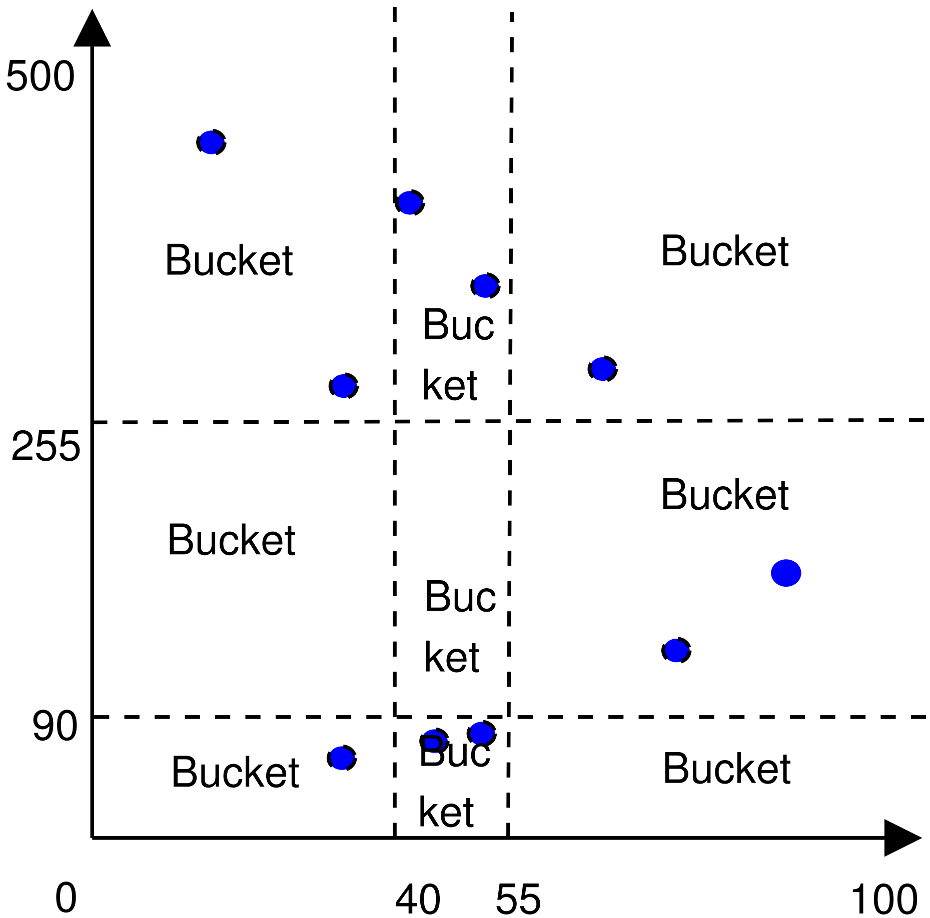
  If the block is full: create overflow blocks or split by creating new separator lines (difficult).

- Delete: find the corresponding bucket, and delete.

  Reorganize if desired

Multidimensional Indexes

Grid files: a variant on hashing



- Good support for point queries
- Good support for partial match queries
- Good support for range queries

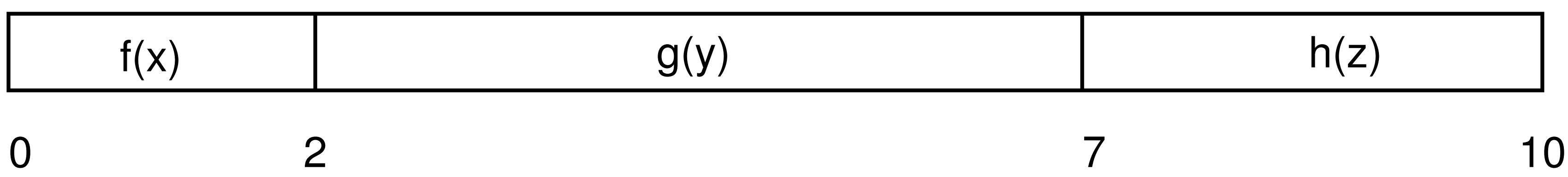
  $\rightarrow$ Lots of buckets to inspect, but also lots of answers
- Reasonable support for nearest-neighbour queries

  $\rightarrow$ By means of neighbourhood searching
- But: many empty buckets when the data is not uniformly distributed

Multidimensional Indexes

Partitioned Hash Functions

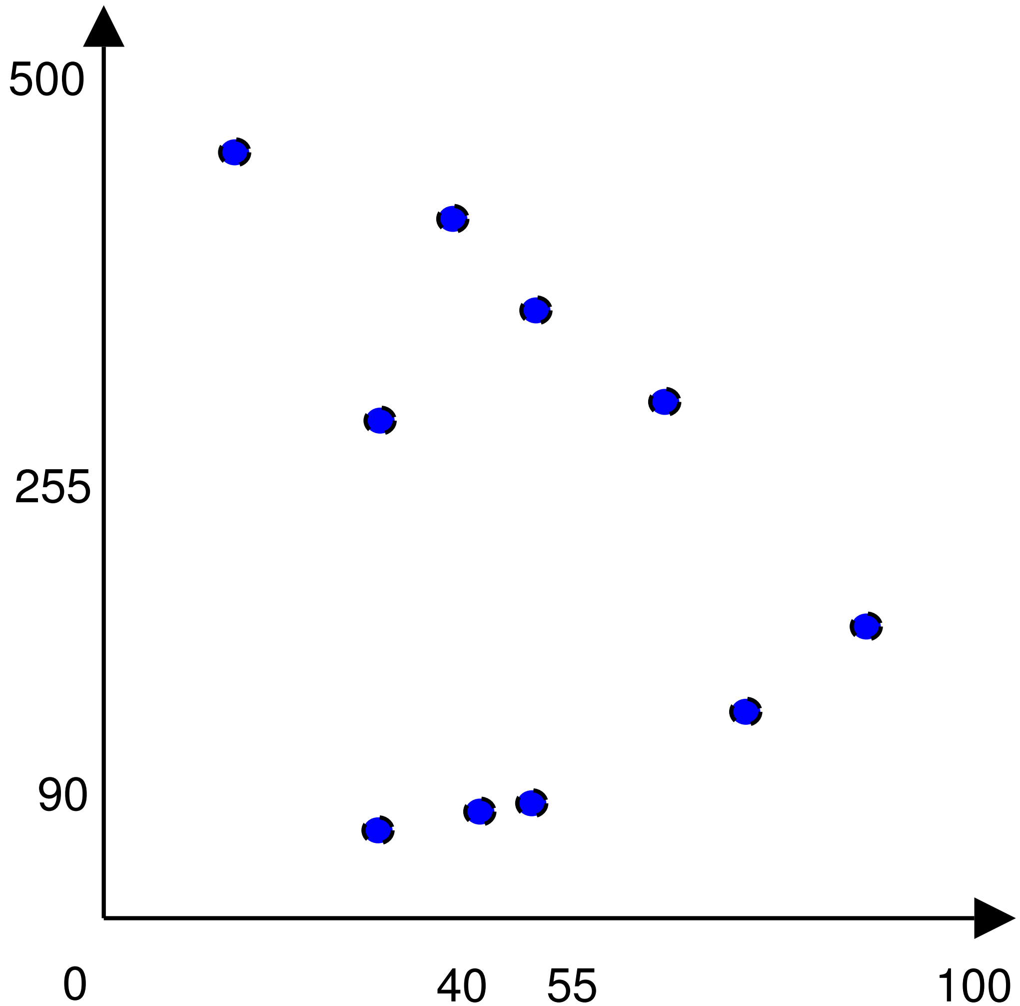
Assume that we have 1024 buckets available to build a hashing index for (x, y, z). We can hence represent each bucket number using 10 bits. Then we can determine the hash value for (x, y, z) as follows:



- Good support for point queries
- Good support for partial match queries
- No support for range queries
- No support for nearest-neighbour queries
- Less wasted space than grid files

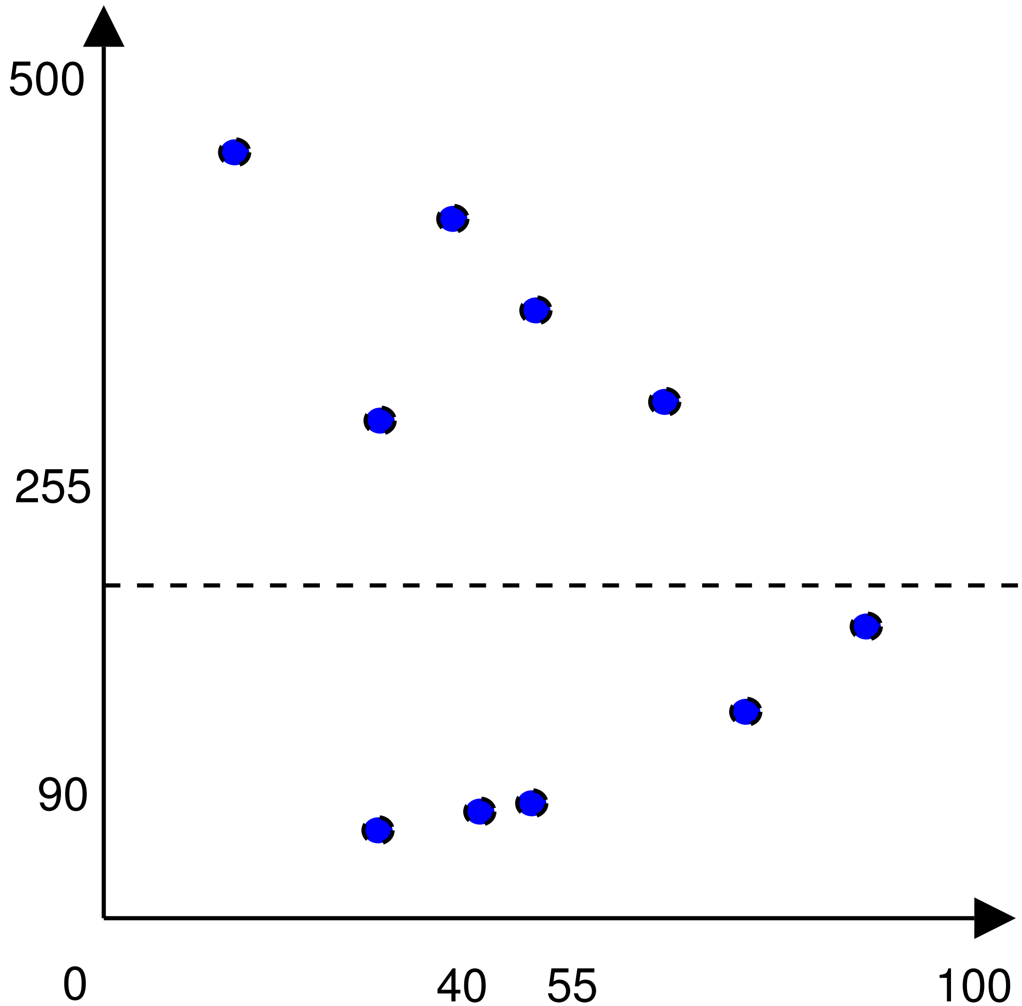
Multidimensional Indexes

kd-Trees



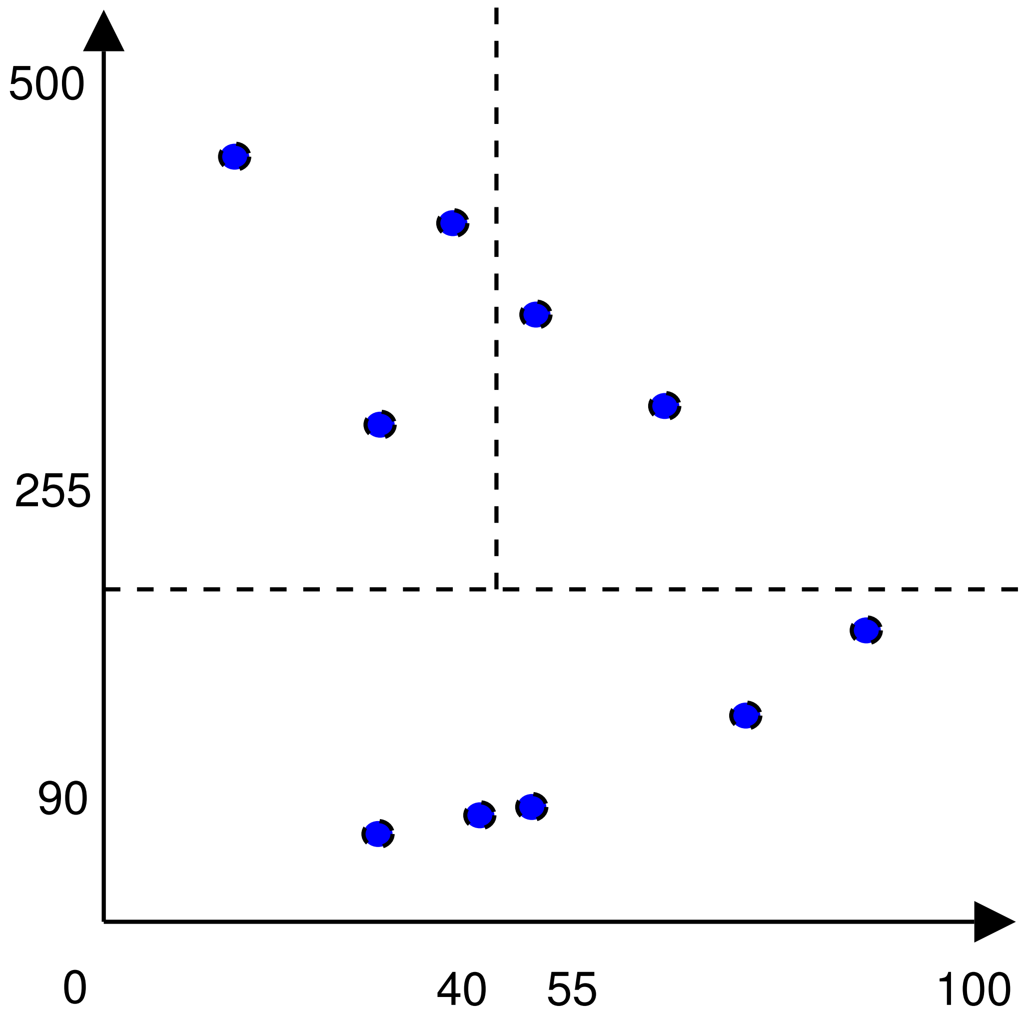
Multidimensional Indexes

kd-Trees



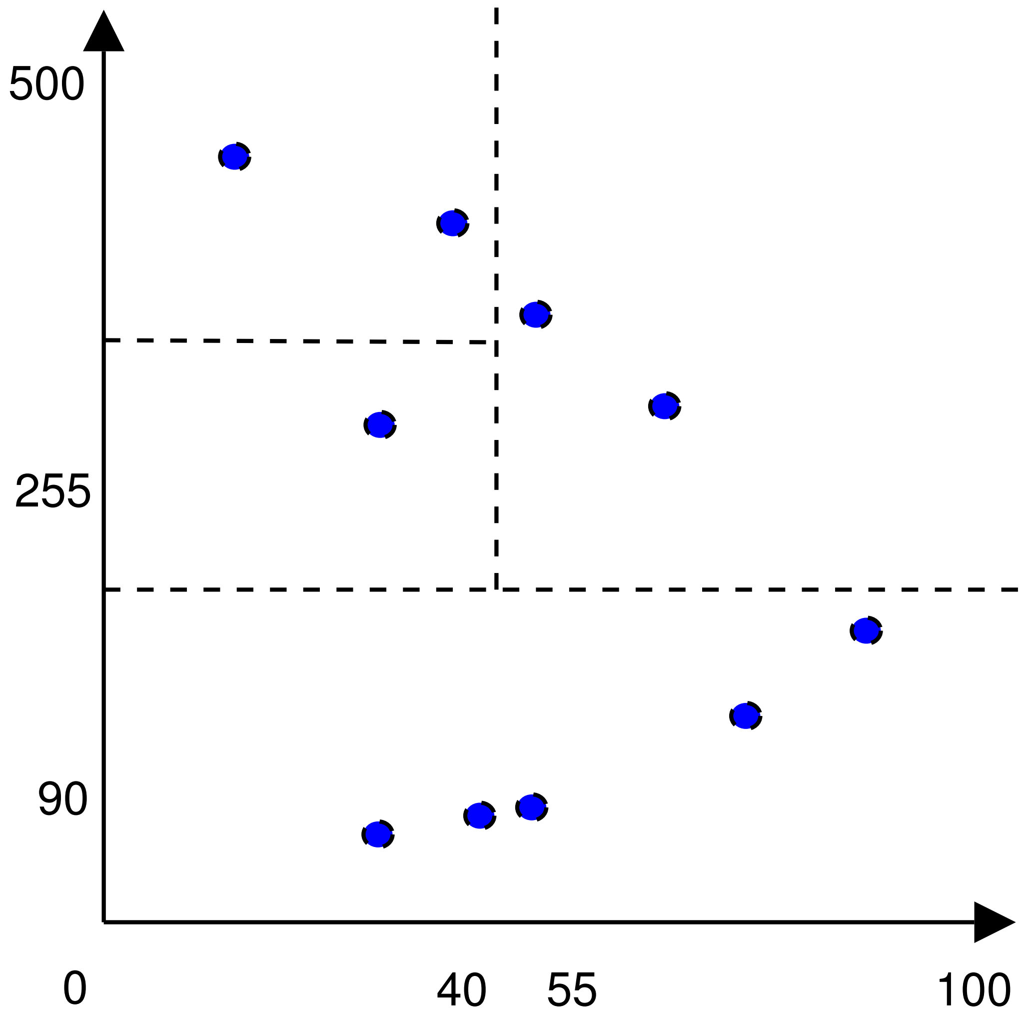
Multidimensional Indexes

kd-Trees



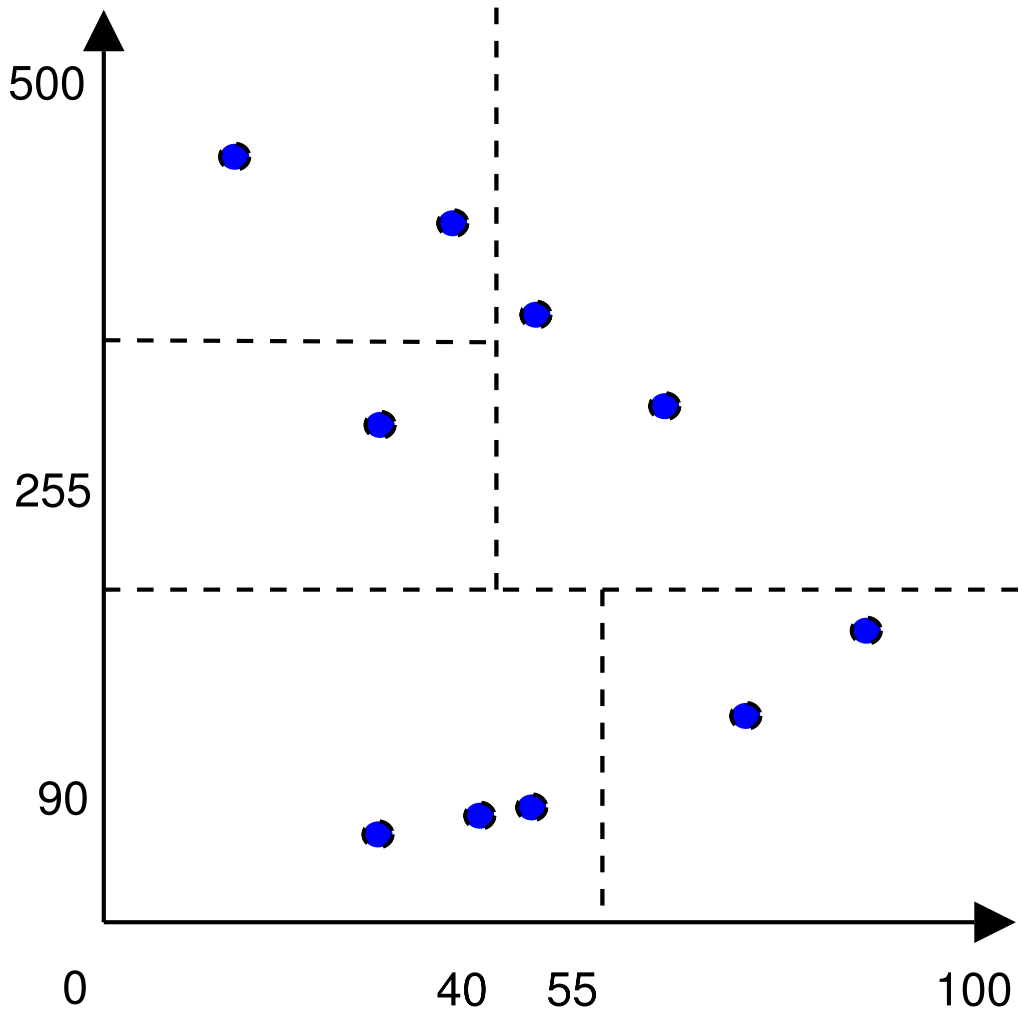
Multidimensional Indexes

kd-Trees



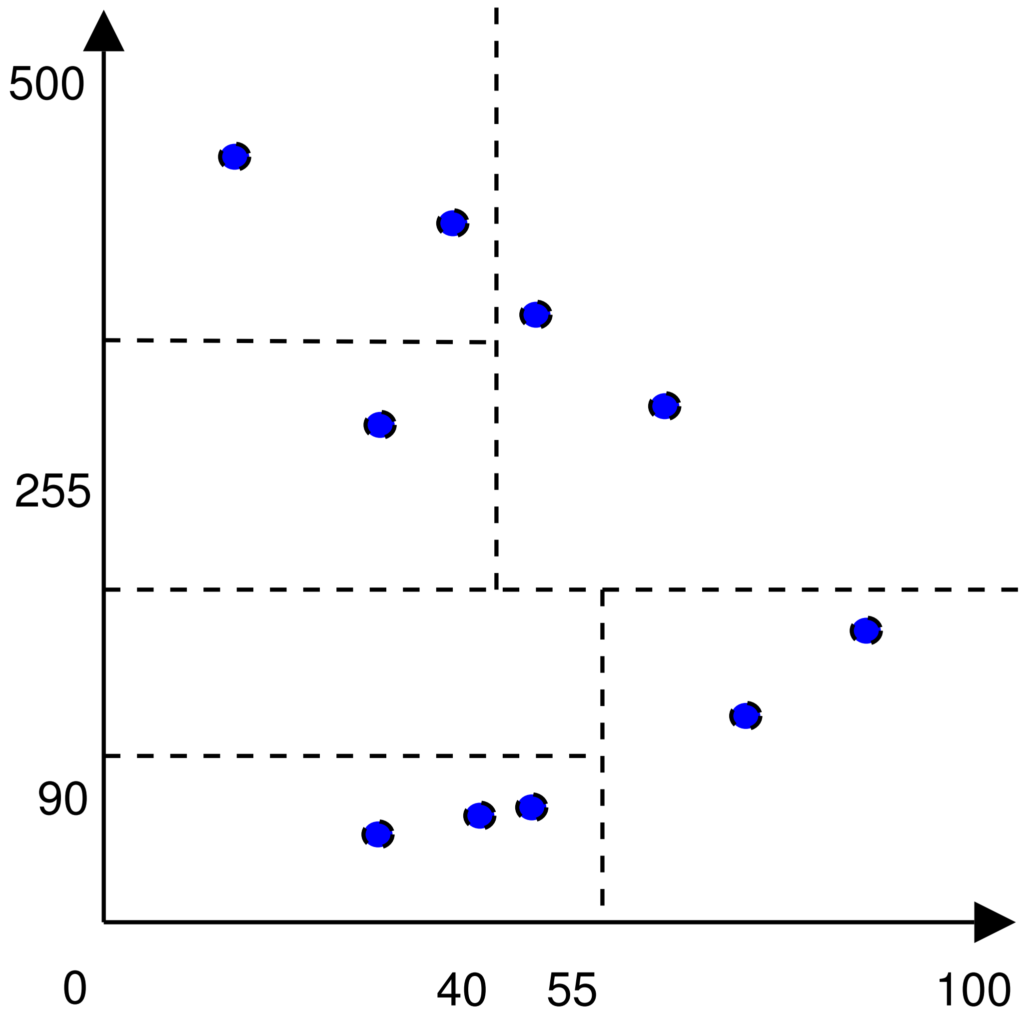
Multidimensional Indexes

kd-Trees



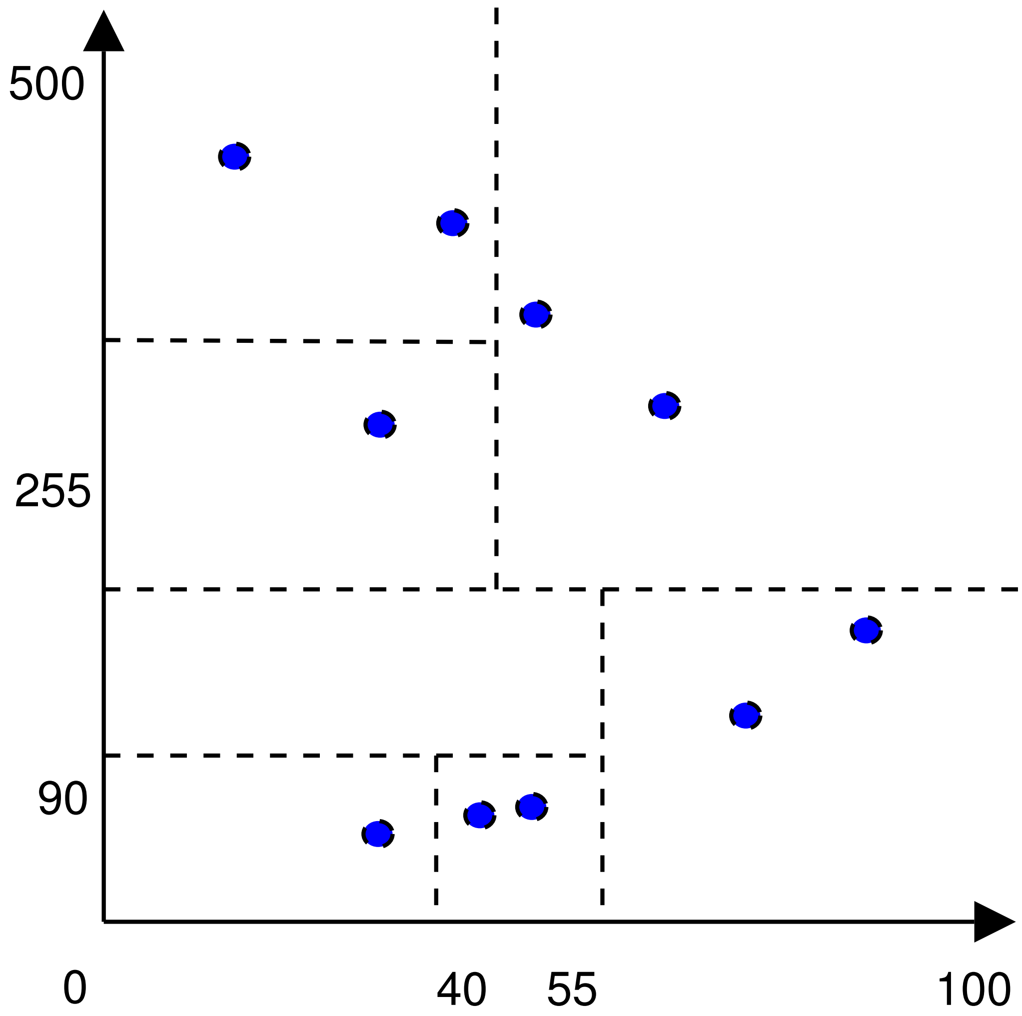
Multidimensional Indexes

kd-Trees



Multidimensional Indexes

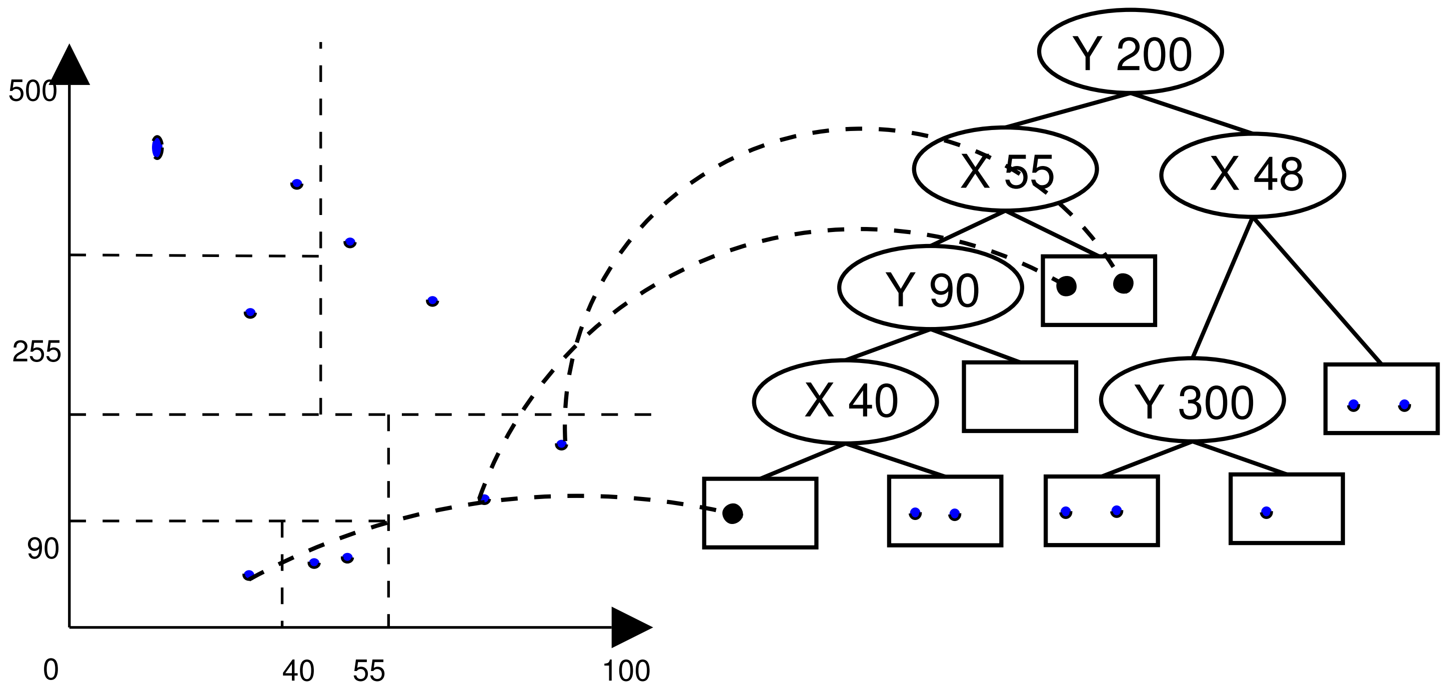
kd-Trees



Multidimensional Indexes

kd-Trees

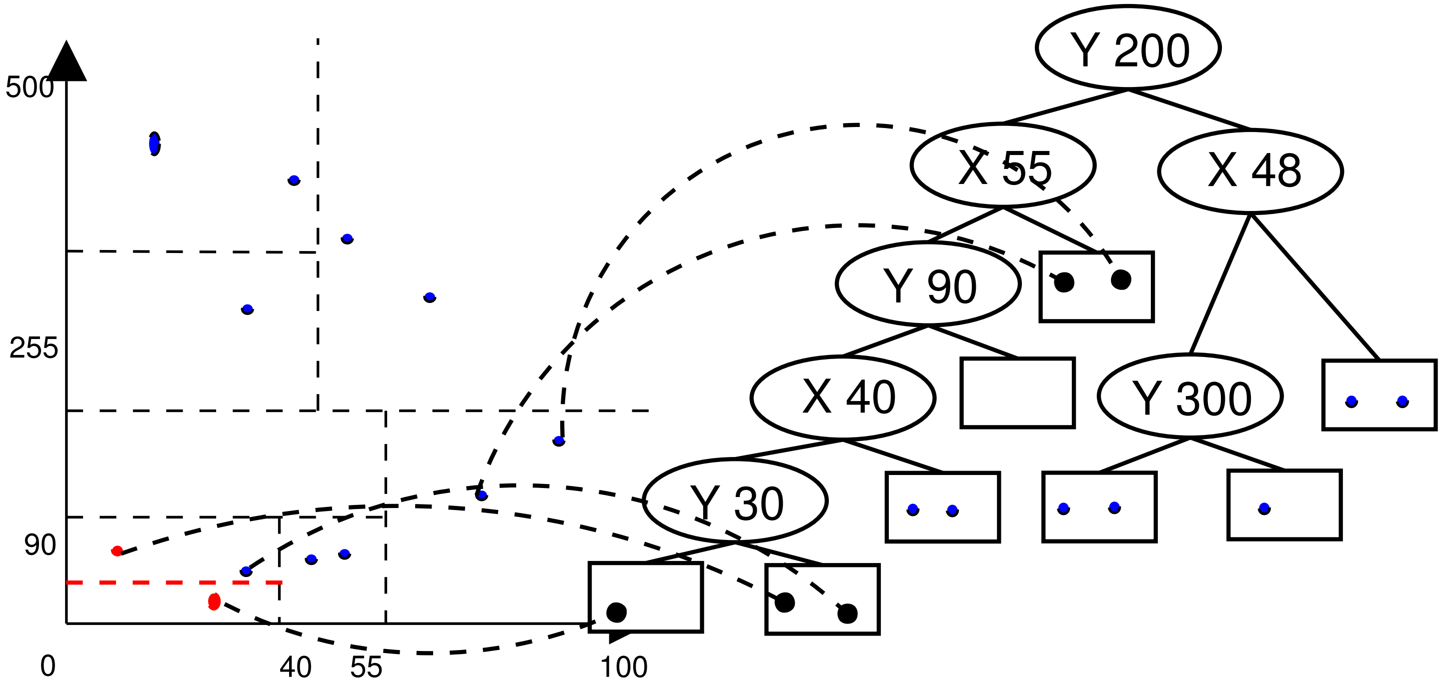
We can look at this as a tree as follows:



Multidimensional Indexes

kd-Trees

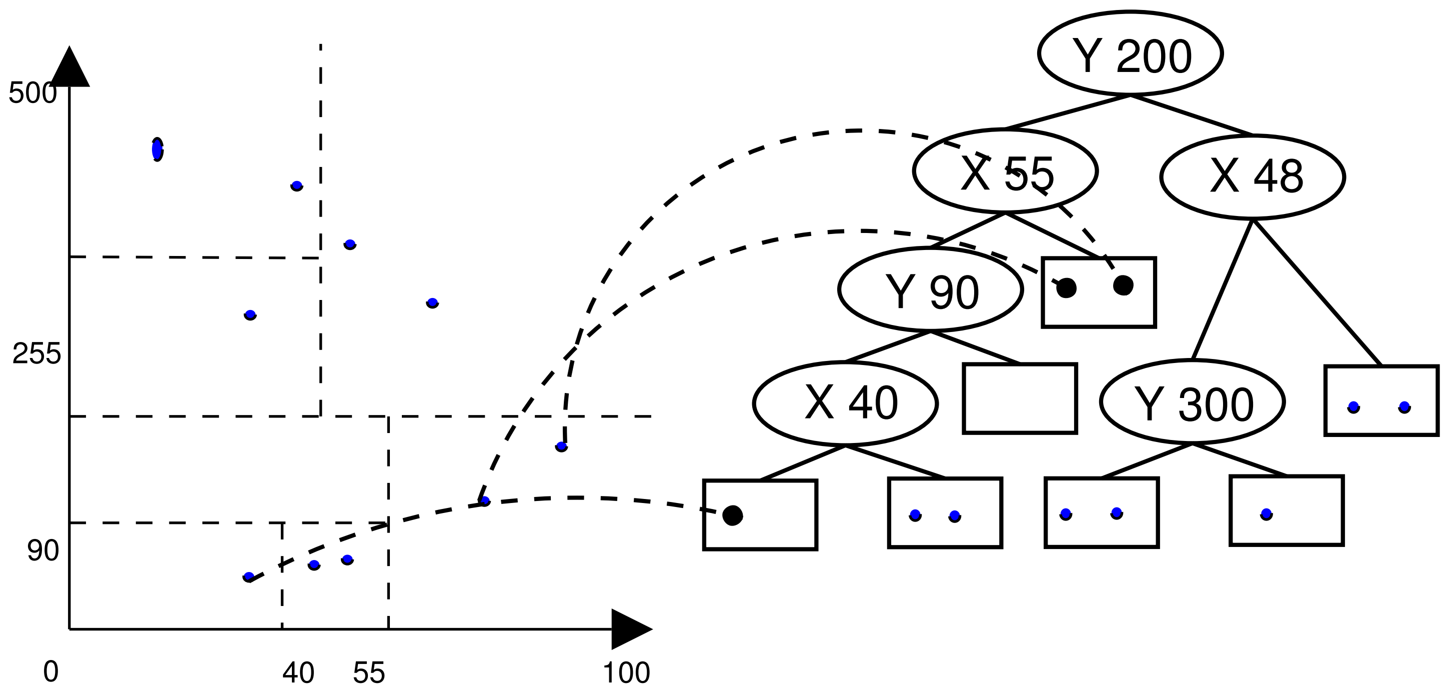
We continue splitting after new insertions:



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).

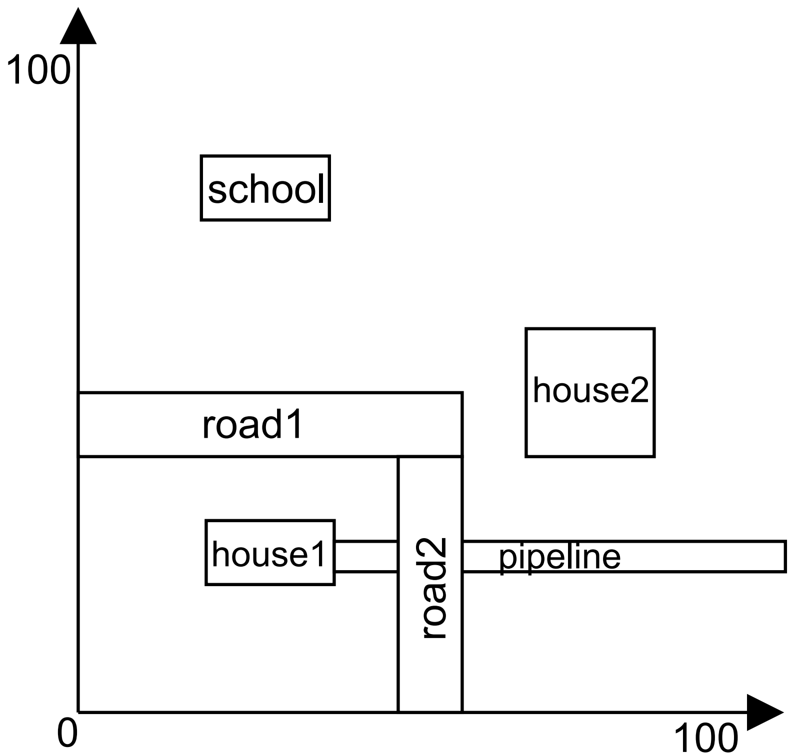
  But it is difficult to keep this tree balanced since we cannot merge the children

- 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). Assume that the following regions fit on a single block:

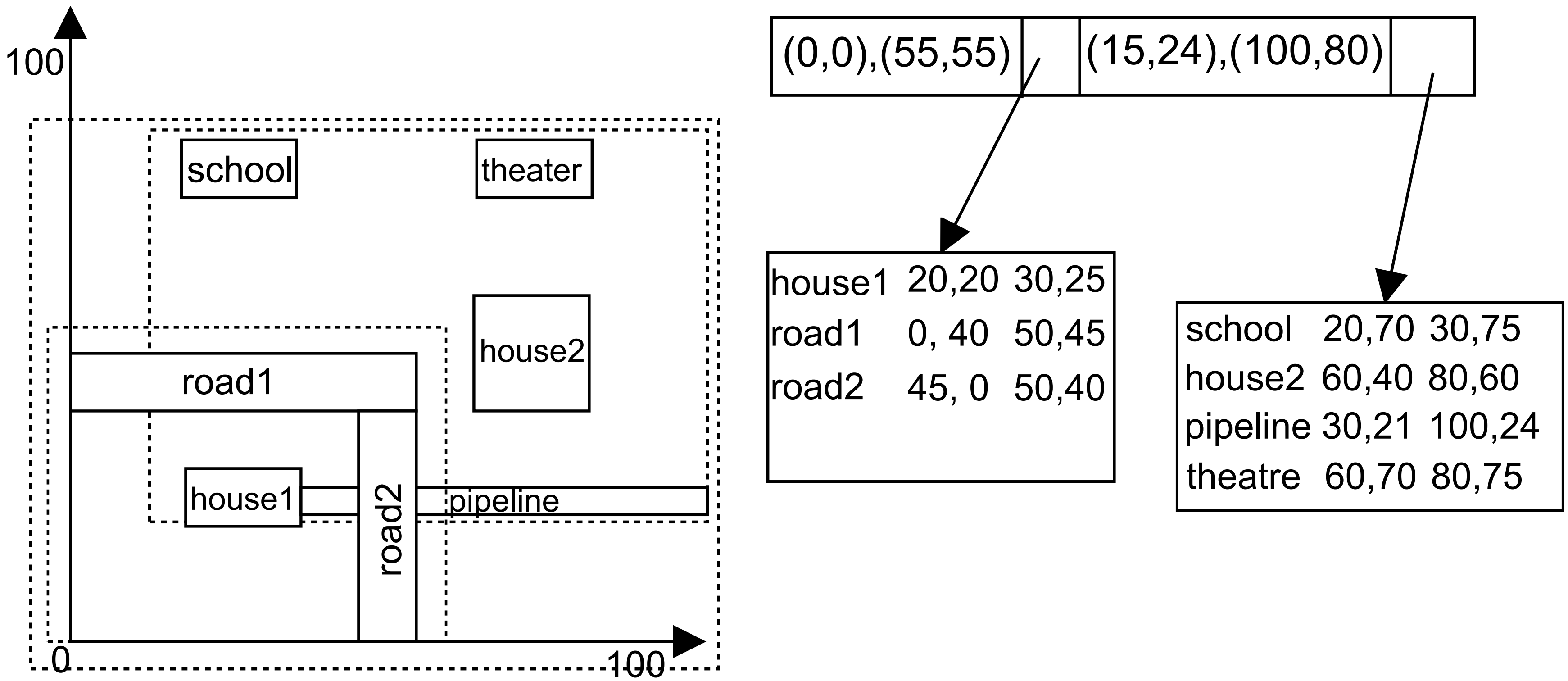


house1	20,20	30,25
road1	0, 40	50,45
road2	45, 0	50,40
school	20,70	30,75
house2	60,40	80,60
pipeline	30,21	100,24

Multidimensional Indexes

R-Trees: generalization of BTrees

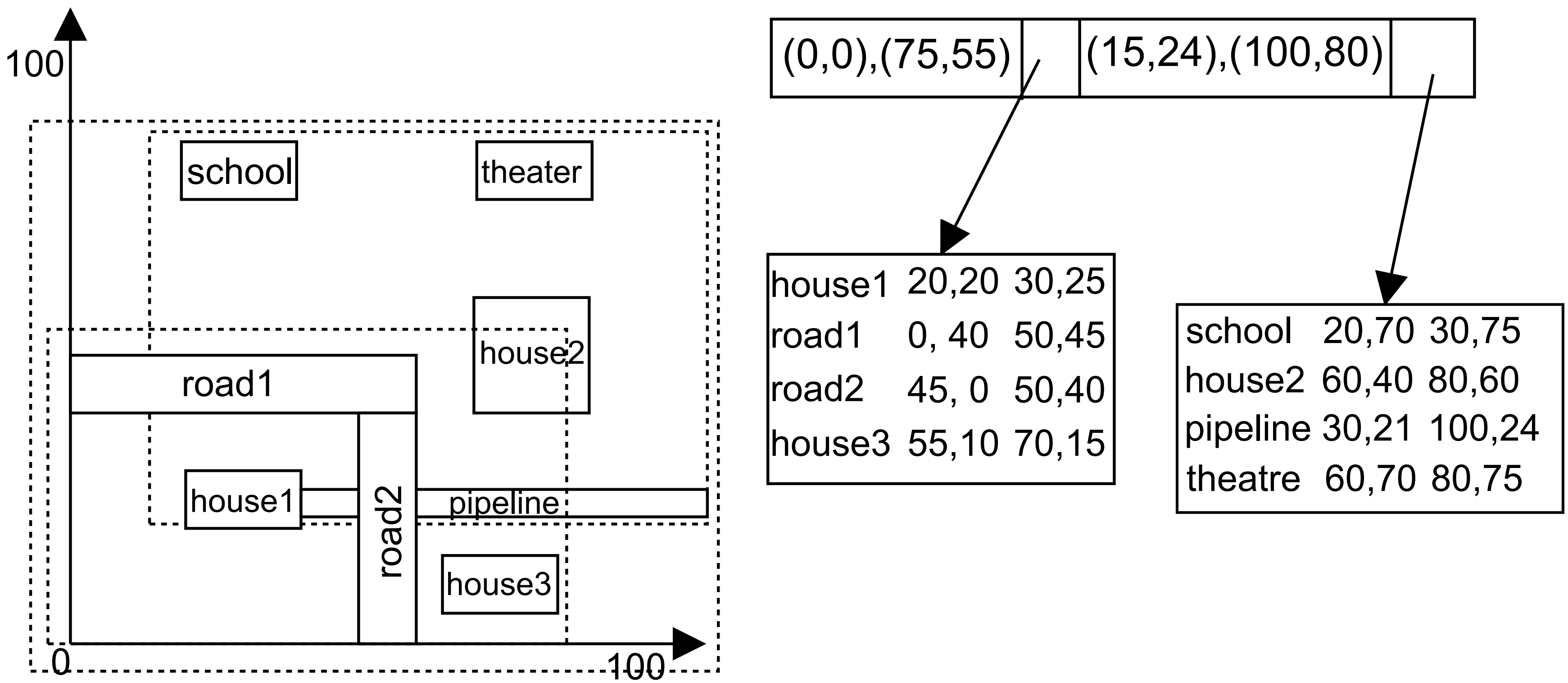
A new region is inserted and we need to split the block into two. We create a tree structure:



Multidimensional Indexes

R-Trees: generalization of BTrees

Inserting again can be done by extending the “bounding regions”:



Multidimensional Indexes

R-Trees: generalization of BTrees

- Ideal for "where-am-I" queries
- Ideal for finding intersecting regions

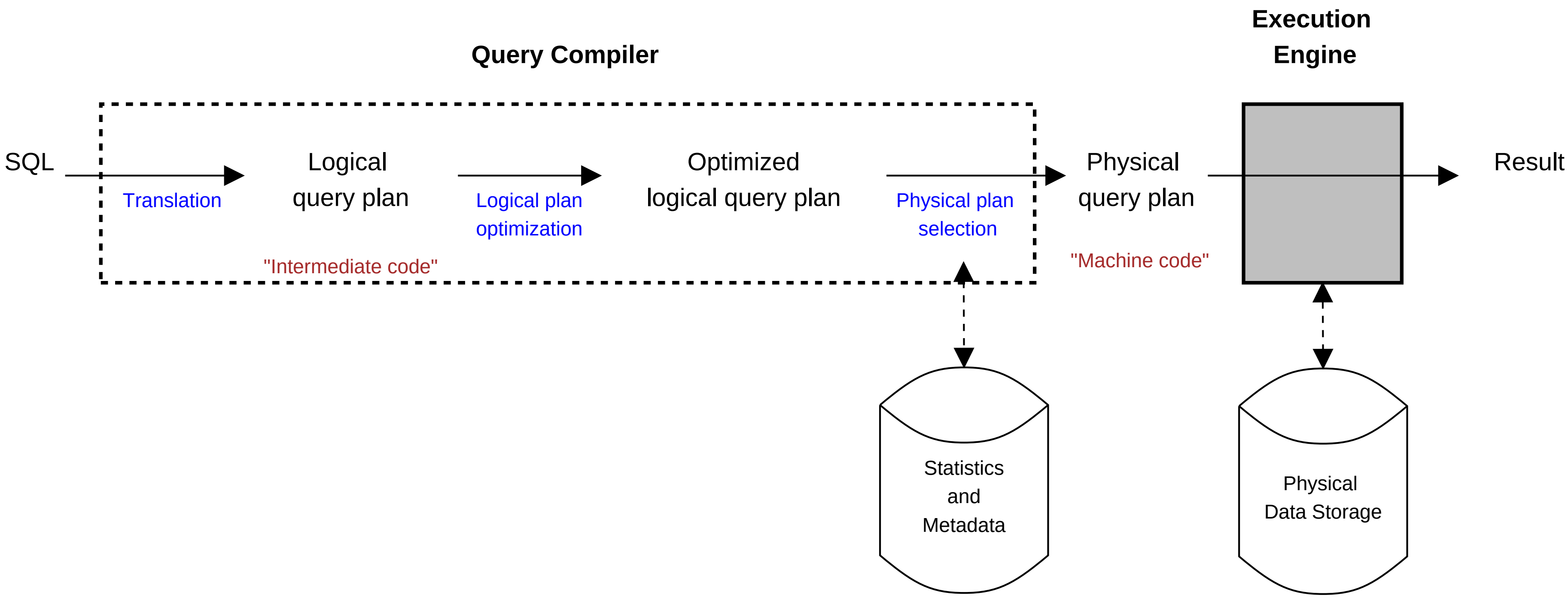
  e.g., when a user highlights an area of interest on a map

- Reasonable support for point queries
- Good support for partial match queries: e.g., $(40 \leq x \leq 45)$
- Good support for range queries
- Reasonable support for nearest neighbour
- Is balanced
- Often used in practice

Physical Operators

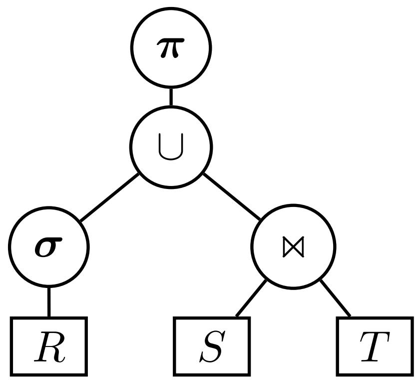
Scanning, sorting, merging, hashing

Physical Operators



Physical Operators

A logical query plan is essentially an execution tree



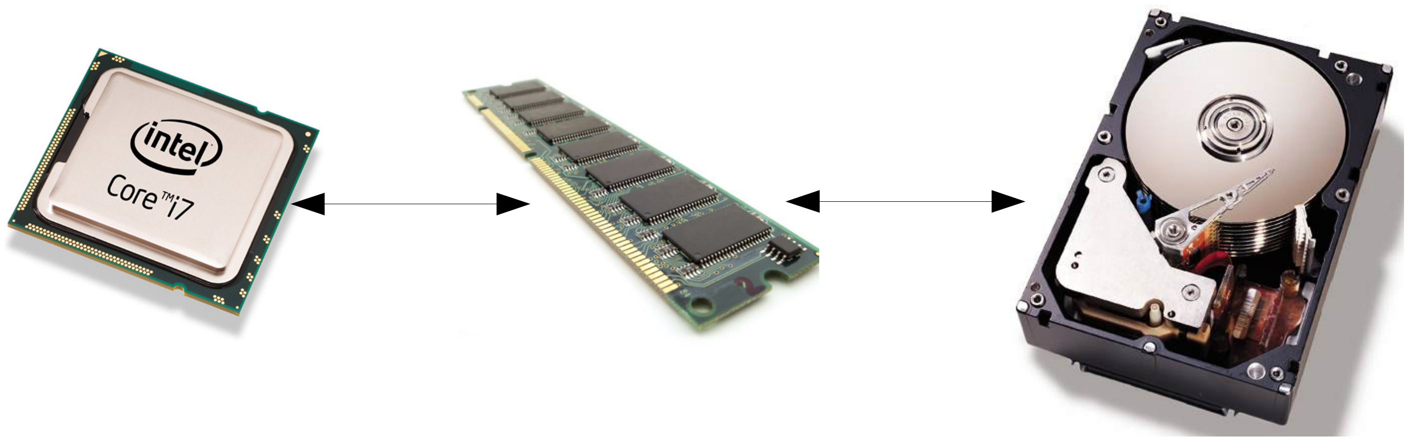
- To obtain a physical query plan we need to assign to each logical operator a physical implementation algorithm. We call such algorithms physical operators.
- In this lecture we study the various physical operators, together with their cost.

Physical Operators

Many implementations

- Each logical operator has multiple possible implementation algorithms
- No implementation is always better the others
- Hence we need to compare the alternatives on a case-by-case basis based on their costs

The I/O model of computation

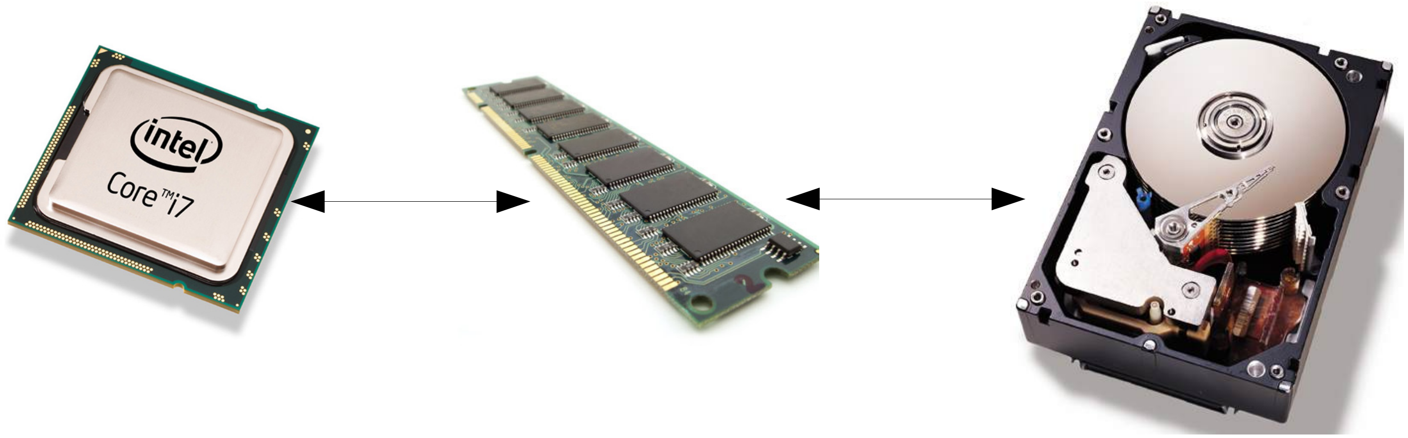


The I/O model

- Data is stored on disk, which is divided into blocks of bytes (typically 4 kilobytes)

  (each block can contain many data items)
- The CPU can only work on data items that are in memory, not on items on disk
- Therefore, data must first be transferred from disk to memory
- Data is transferred from disk to memory (and back) in whole blocks at the time
- The disk can hold D blocks, at most M blocks can be in memory at the same time (with $M << D$).

The I/O model of computation



- In-memory computation is fast (memory access $\approx 10^{-8}s$)
- Disk-access is slow (disk access: $\approx 10^{-3}s$)
- Hence: execution time is dominated by disk I/O

We will use the number of I/O operations required as cost metric

Physical Operators

To estimate the costs we will use the following parameters:

- $B(R)$: the number of blocks that R occupies on disk
- $T(R)$: the number of tuples in relation R
- $V(R, A_1, \ldots, A_n)$: the number of tuples in R that have distinct values for $A_1, \ldots, A_n$

  (i.e., $|\delta(\pi_{A_1,\ldots,A_n}(R)|$)
- M: the number of main memory buffers available

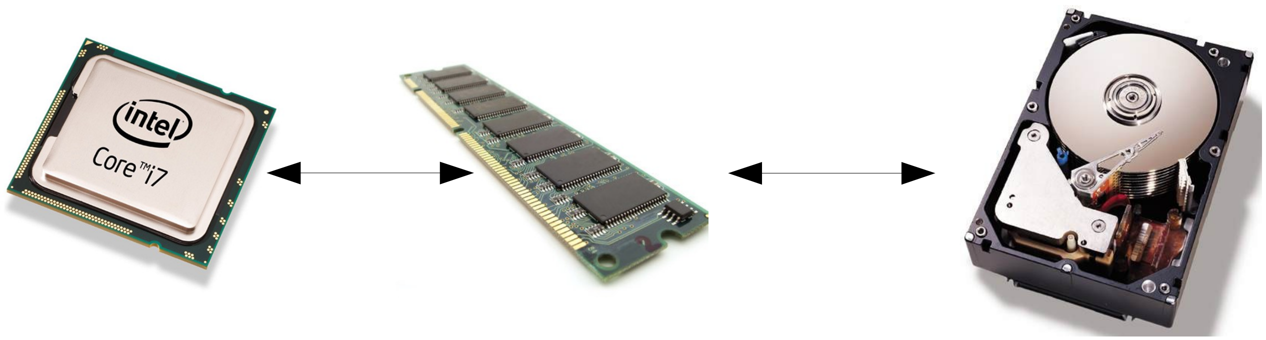
Statistics and the system catalog

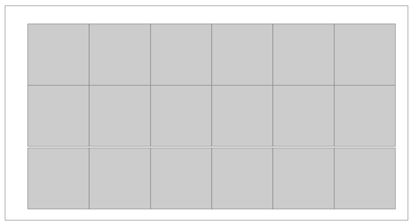
- The first three parameters are statistics that a DBMS stores in its system catalog
- These statistics are regularly collected

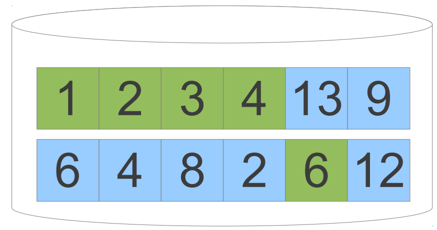
  (e.g., when required, at a scheduled time, ...)

Physical Operators

Bag union $R \cup_B S$



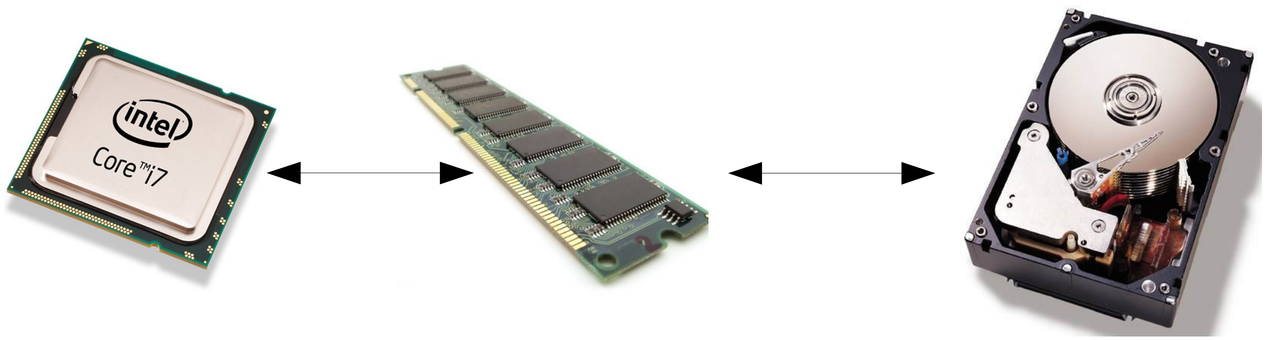


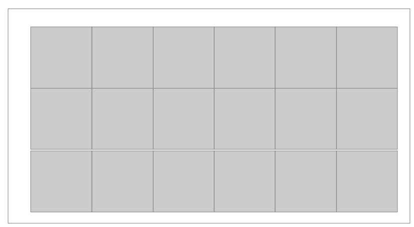


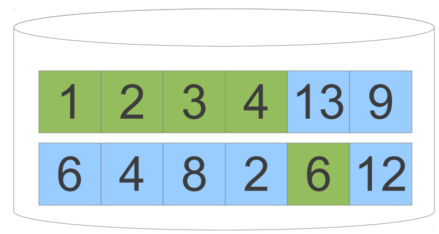
Relation R = green
Relation S = blue
1 integer per block

Physical Operators

Bag union $R \cup_B S$





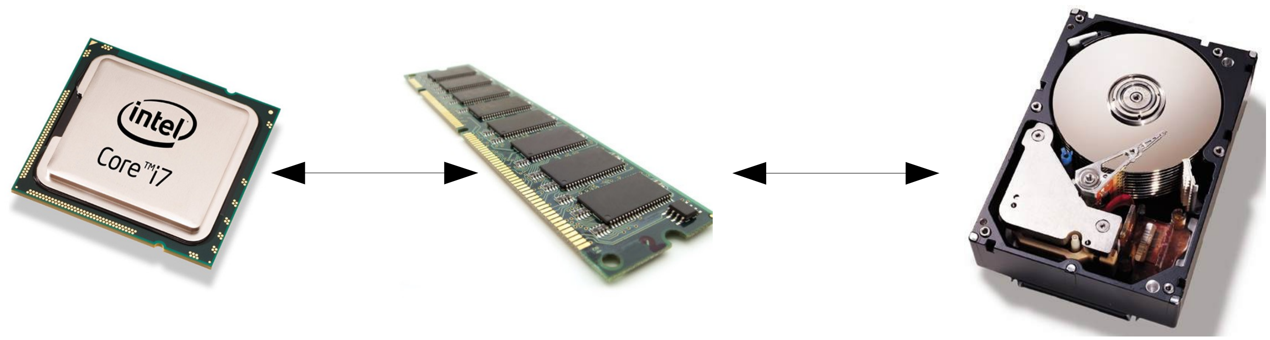


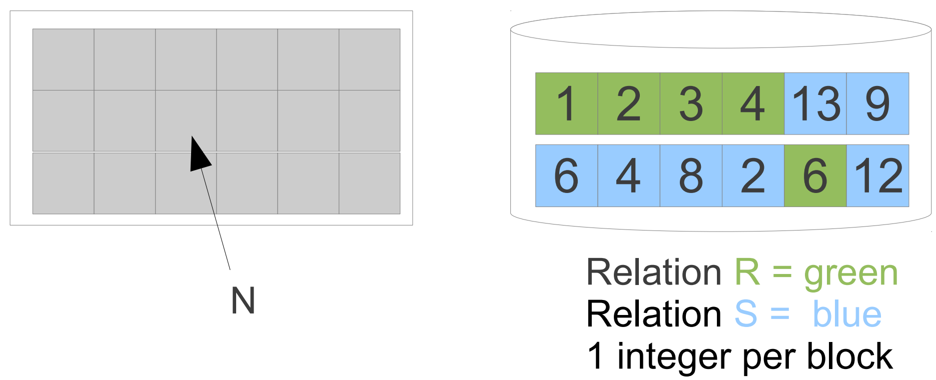
Relation R = green
Relation S = blue
1 integer per block

- Step 1: reserve 1 buffer frame, call this N

Physical Operators

Bag union $R \cup_B S$

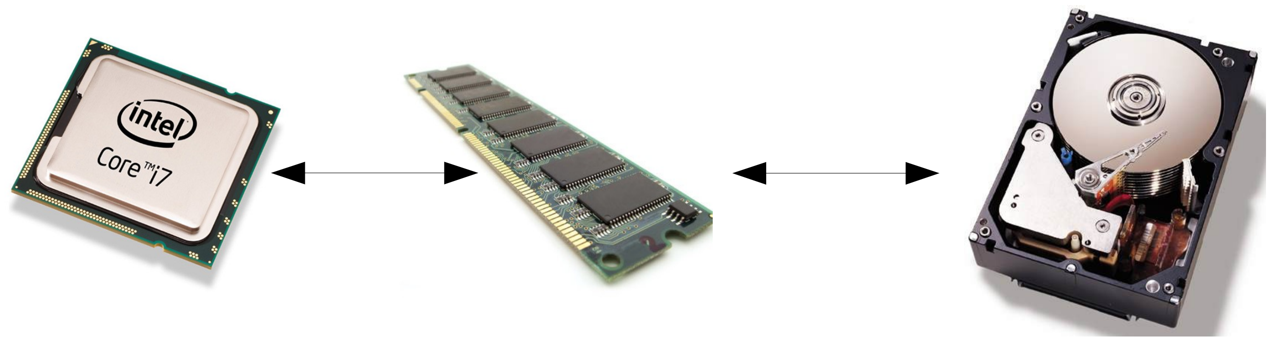


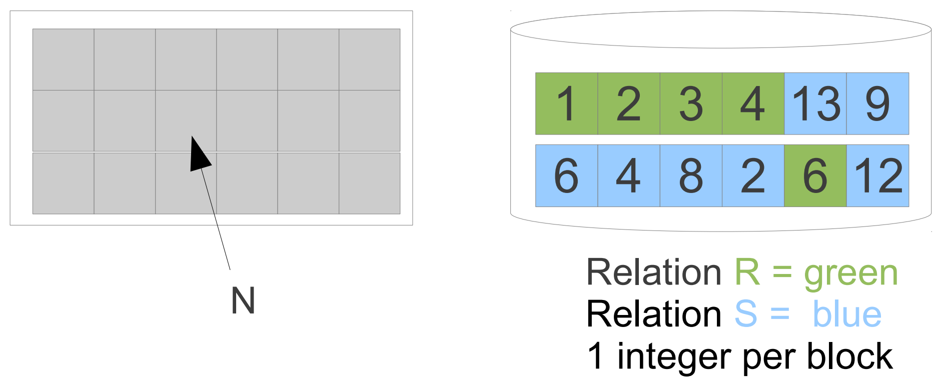


- Step 1: reserve 1 buffer frame, call this N

Physical Operators

Bag union $R \cup_B S$

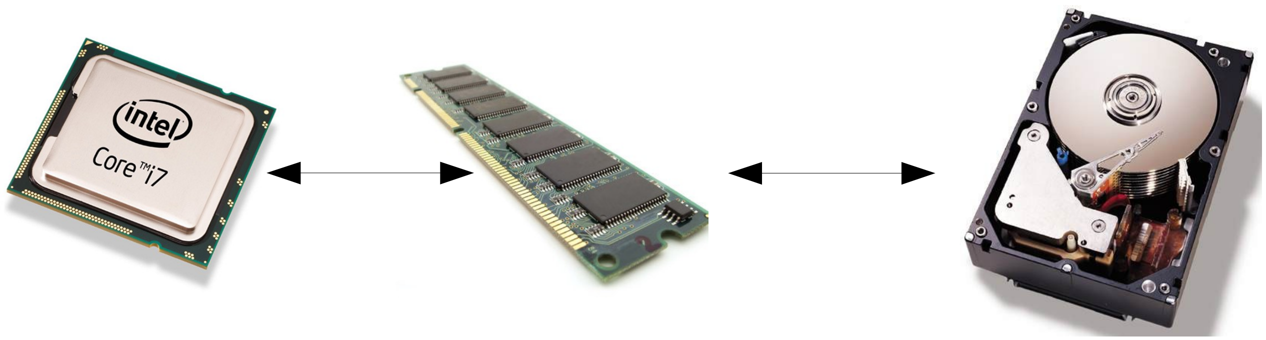


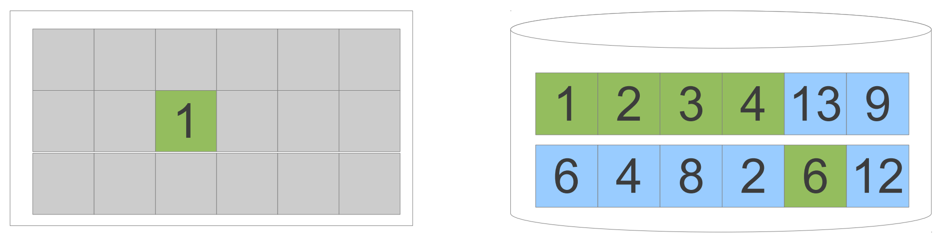


- Load 1st block of R into N, output all of its elements

Physical Operators

Bag union $R \cup_B S$



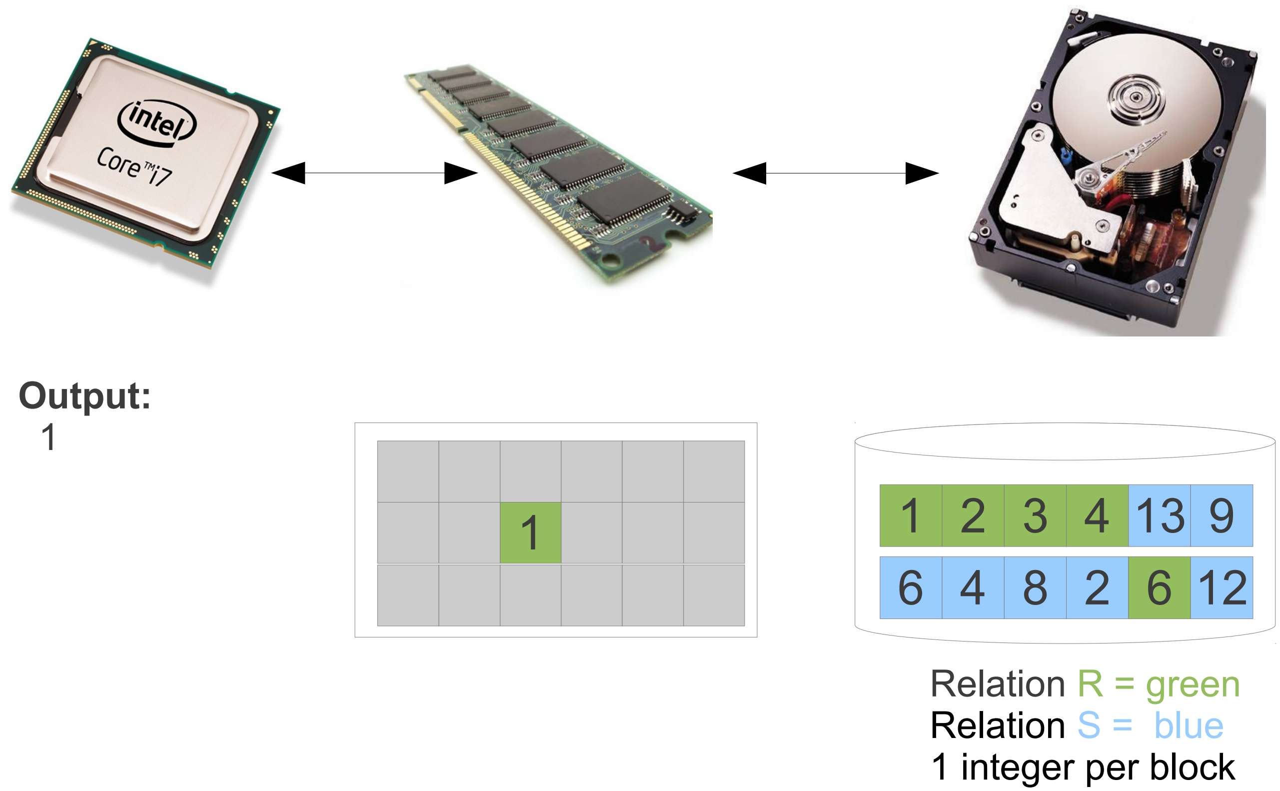


Relation R = green
Relation S = blue
1 integer per block

- Load 1st block of R into N, output all of its elements

Physical Operators

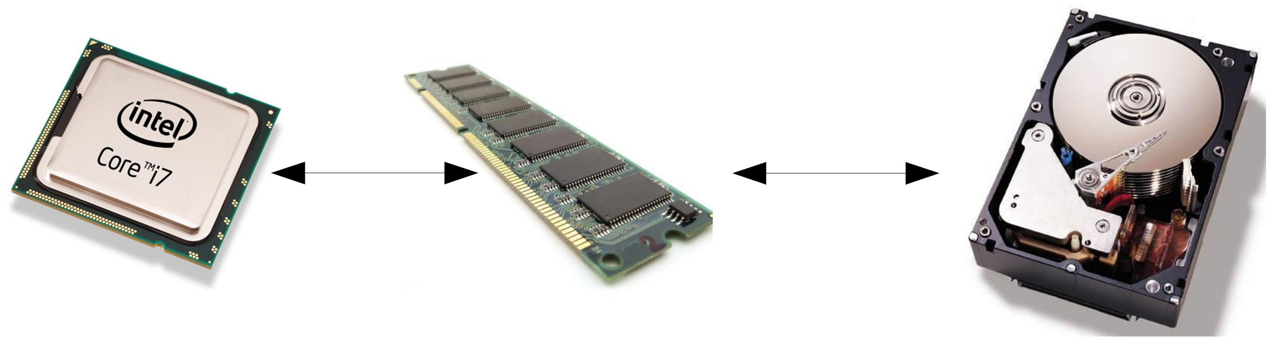
Bag union $R \cup_B S$



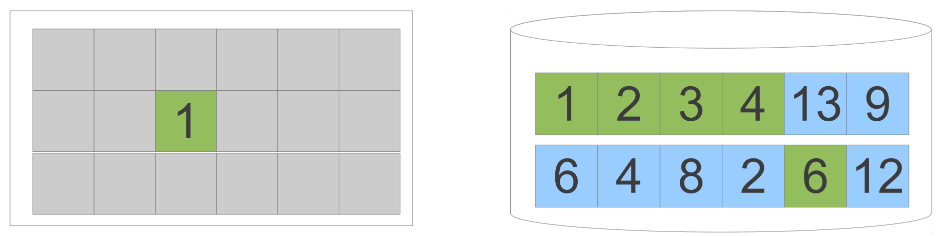
- Load 1st block of R into N, output all of its elements

Physical Operators

Bag union $R \cup_B S$



Output:
1

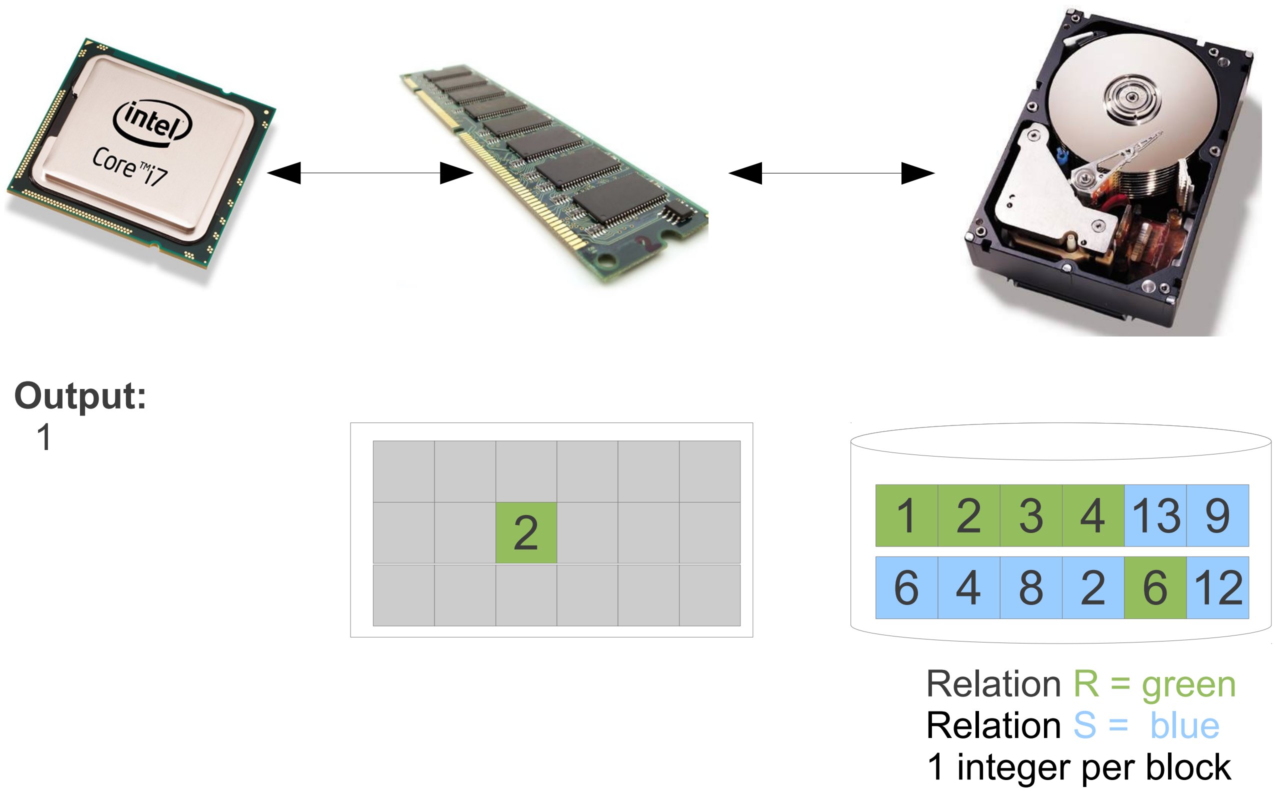


Relation R = green
Relation S = blue
1 integer per block

- Load 2nd block of R into N, output all of its elements

Physical Operators

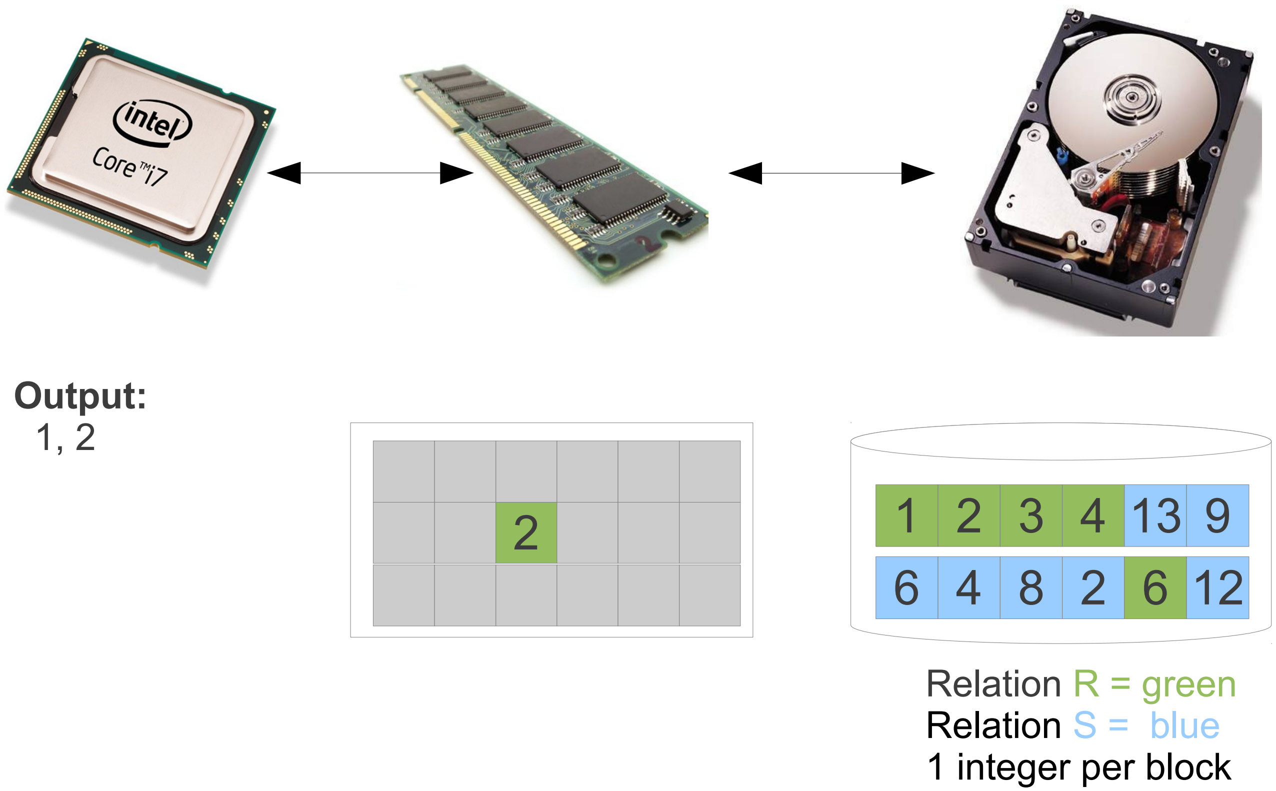
Bag union $R \cup_B S$



- Load 2nd block of R into N, output all of its elements

Physical Operators

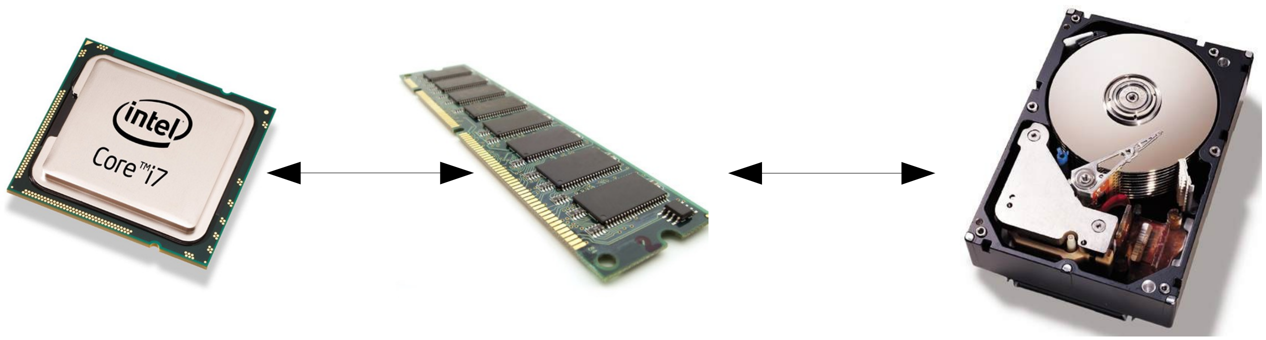
Bag union $R \cup_B S$



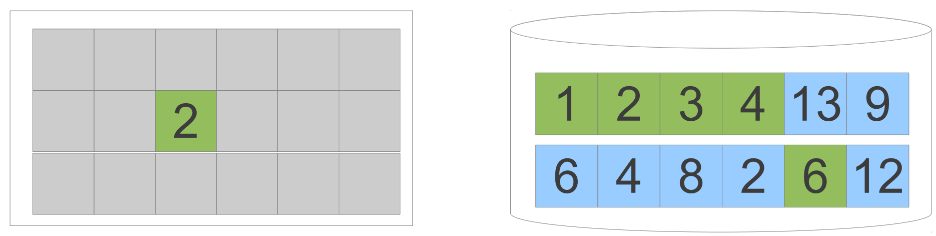
- Load 2nd block of R into N, output all of its elements

Physical Operators

Bag union $R \cup_B S$



Output:
1, 2

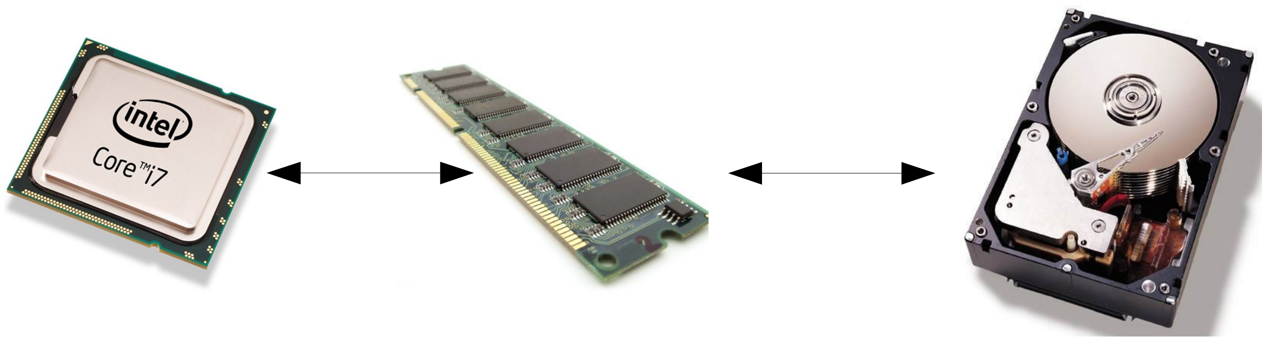


Relation R = green
Relation S = blue
1 integer per block

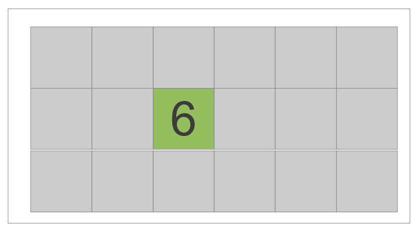
- ...and repeat this for every block of R

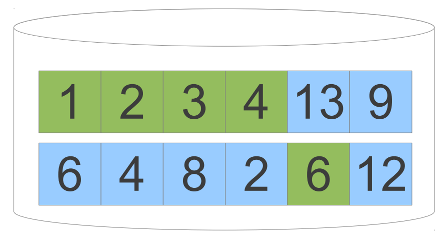
Physical Operators

Bag union $R \cup_B S$



Output:
1, 2, 3, 4, 6



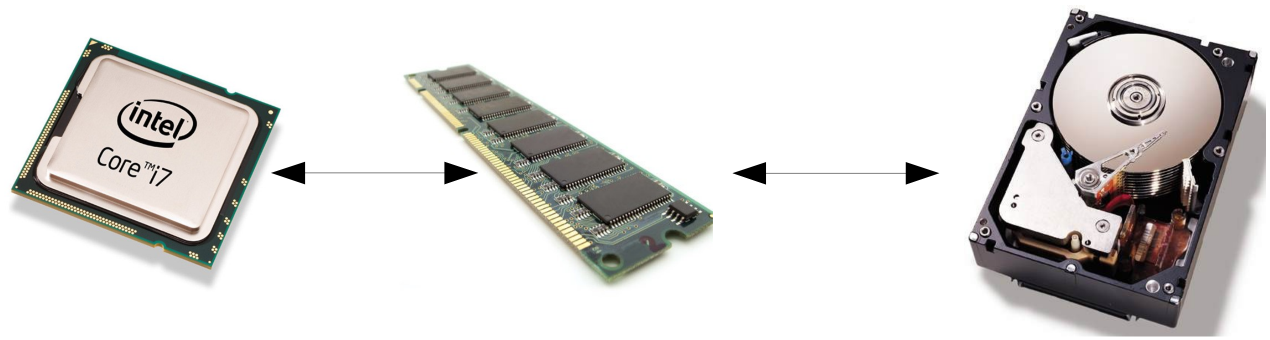


Relation R = green
Relation S = blue
1 integer per block

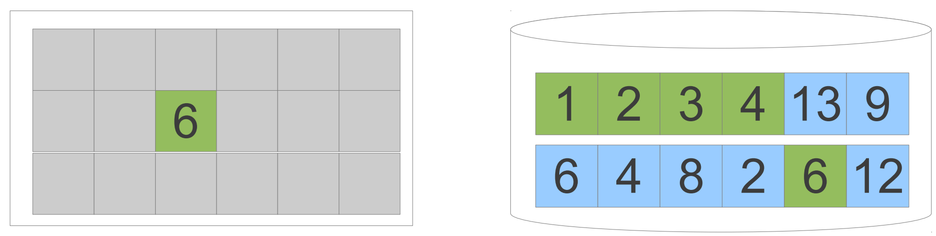
- . . . and repeat this for every block of R.

Physical Operators

Bag union $R \cup_B S$



Output:
1, 2, 3, 4, 6

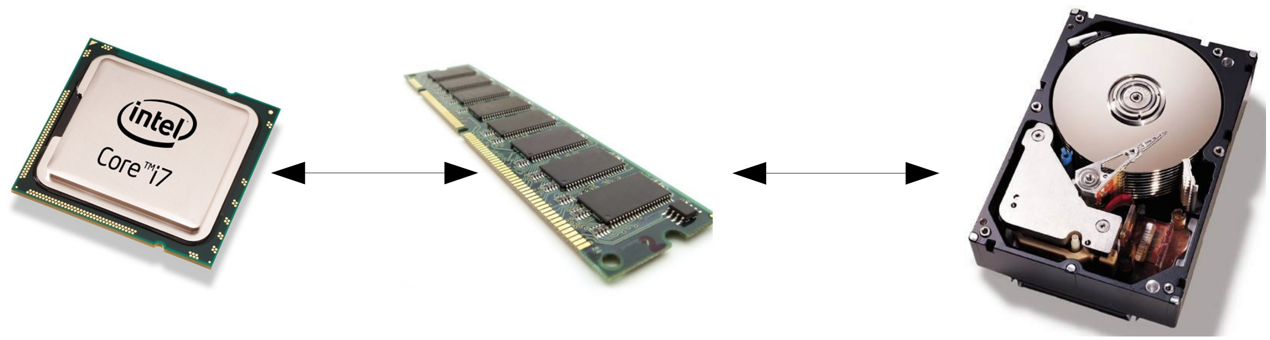


Relation R = green
Relation S = blue
1 integer per block

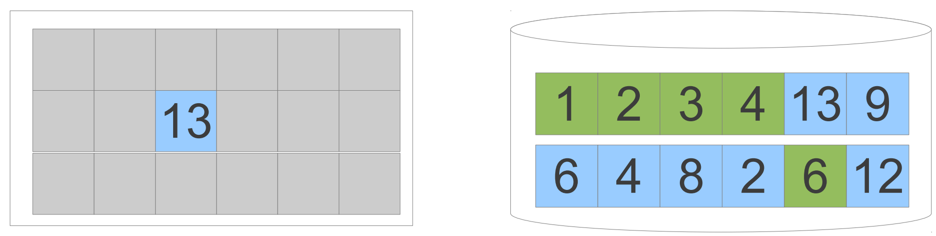
- Load 1st block of S into N, output all of its elements

Physical Operators

Bag union $R \cup_B S$



Output:
1, 2, 3, 4, 6

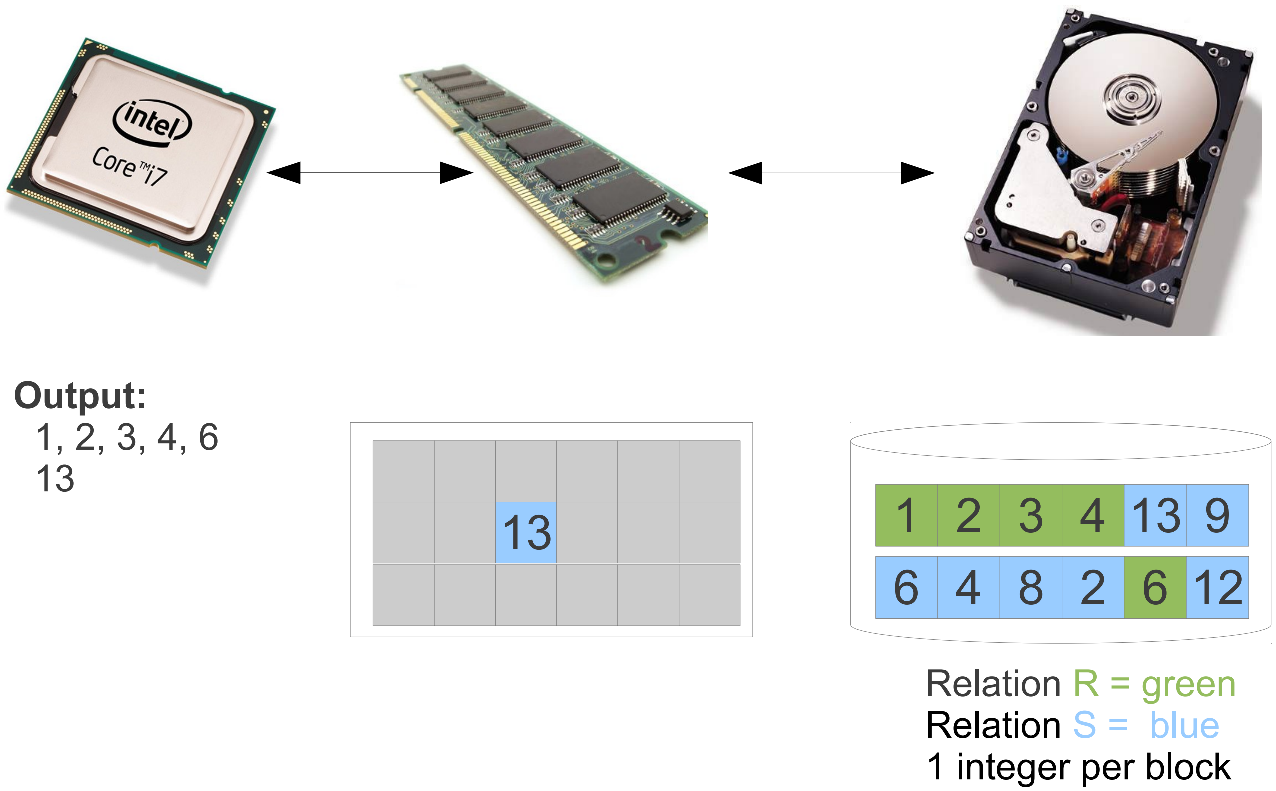


Relation R = green
Relation S = blue
1 integer per block

- Load 1st block of S into N, output all of its elements

Physical Operators

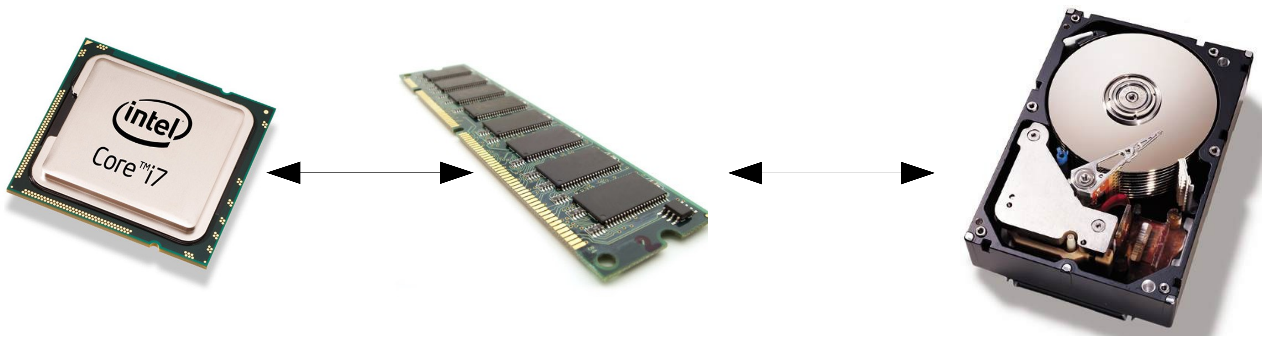
Bag union $R \cup_B S$



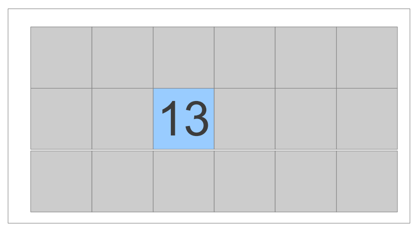
- Load 1st block of S into N, output all of its elements

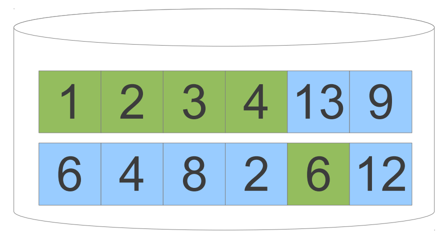
Physical Operators

Bag union $R \cup_B S$



Output:
1, 2, 3, 4, 6
13



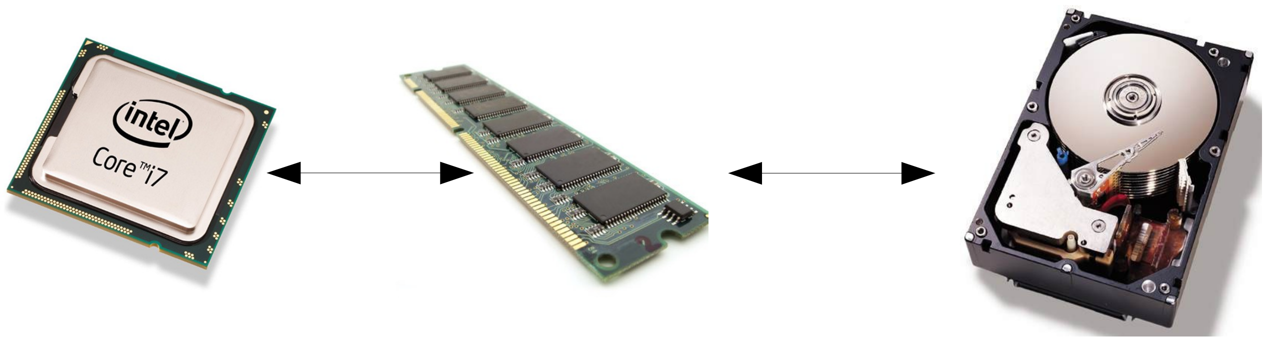


Relation R = green
Relation S = blue
1 integer per block

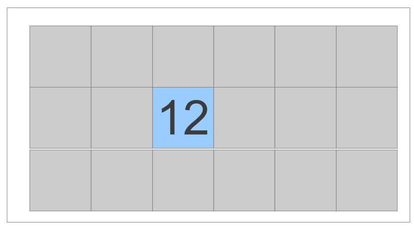
- . . . and repeat this until the last block of S

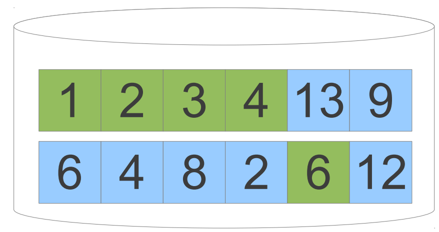
Physical Operators

Bag union $R \cup_B S$



Output:
1, 2, 3, 4, 6
13, 9, 6, 4, 8,
2, 12





Relation R = green
Relation S = blue
1 integer per block

- . . . and repeat this until the last block of S

Physical Operators

Bag union

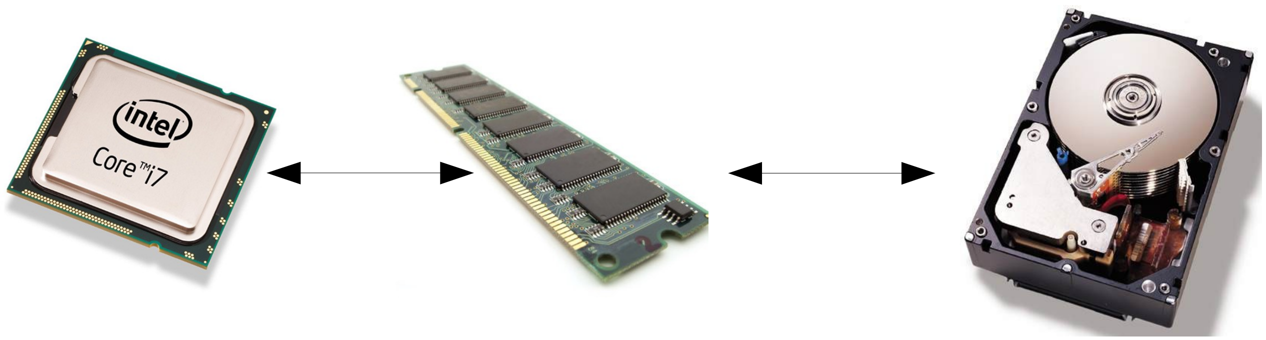
We can compute the bag union $R \cup_B S$ as follows:

for each block B_R in R **do**
  load B_R into buffer N;
  for each tuple t_R in N **do**
    output t_R;
for each block B_S in S **do**
  load B_S into buffer N;
  for each tuple t_S in N **do**
    output t_S;

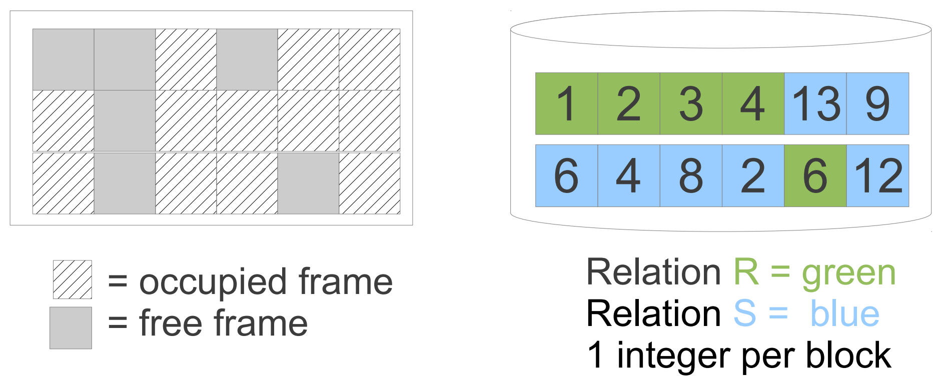
- Cost: $B(R) + B(S)$ I/O operations (we never count the output-cost)
- Requires that $M \geq 1$ (i.e., it can always be used)

Physical Operators

One-pass set union $R \cup_S S$



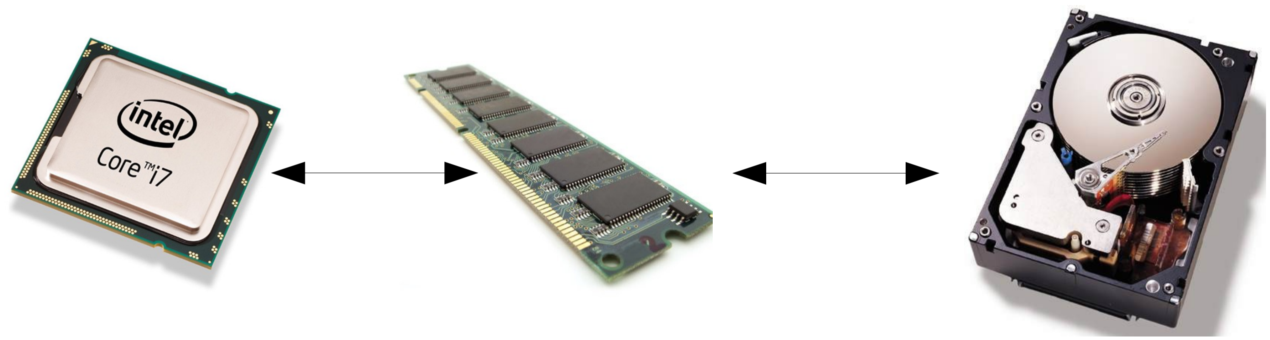
Output:



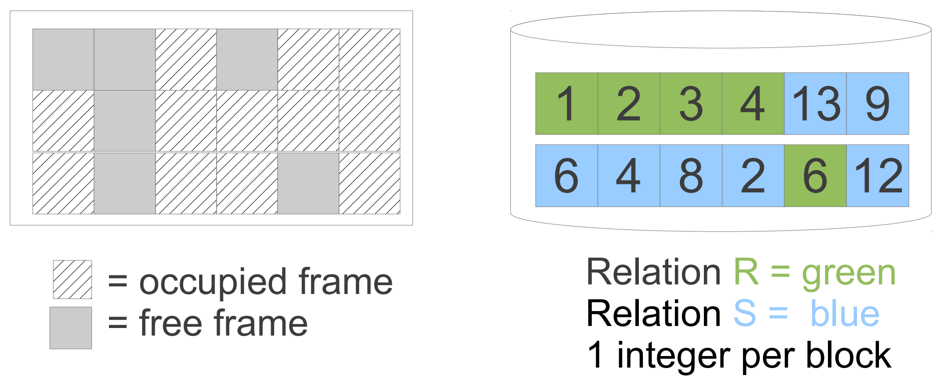
Assumption: we have $B(R) + 1$ free buffer frames

Physical Operators

One-pass set union $R \cup_S S$



Output:

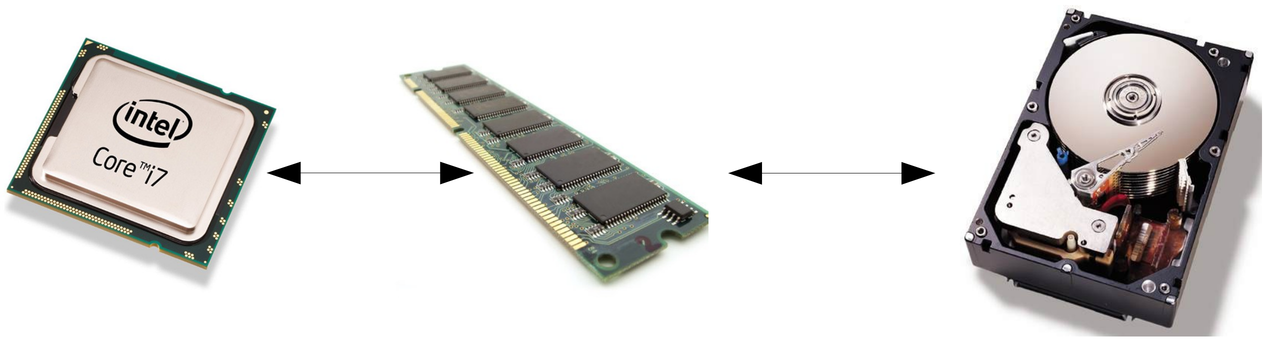


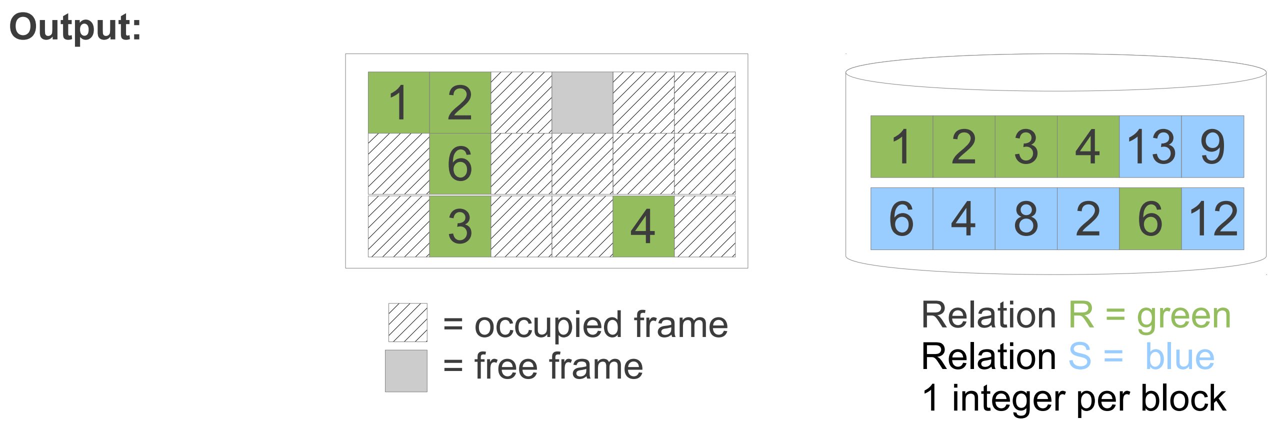
Assumption: we have $B(R) + 1$ free buffer frames

- Load all of R's blocks into memory (using $B(R)$ buffer frames) and output their elements.

Physical Operators

One-pass set union $R \cup_S S$



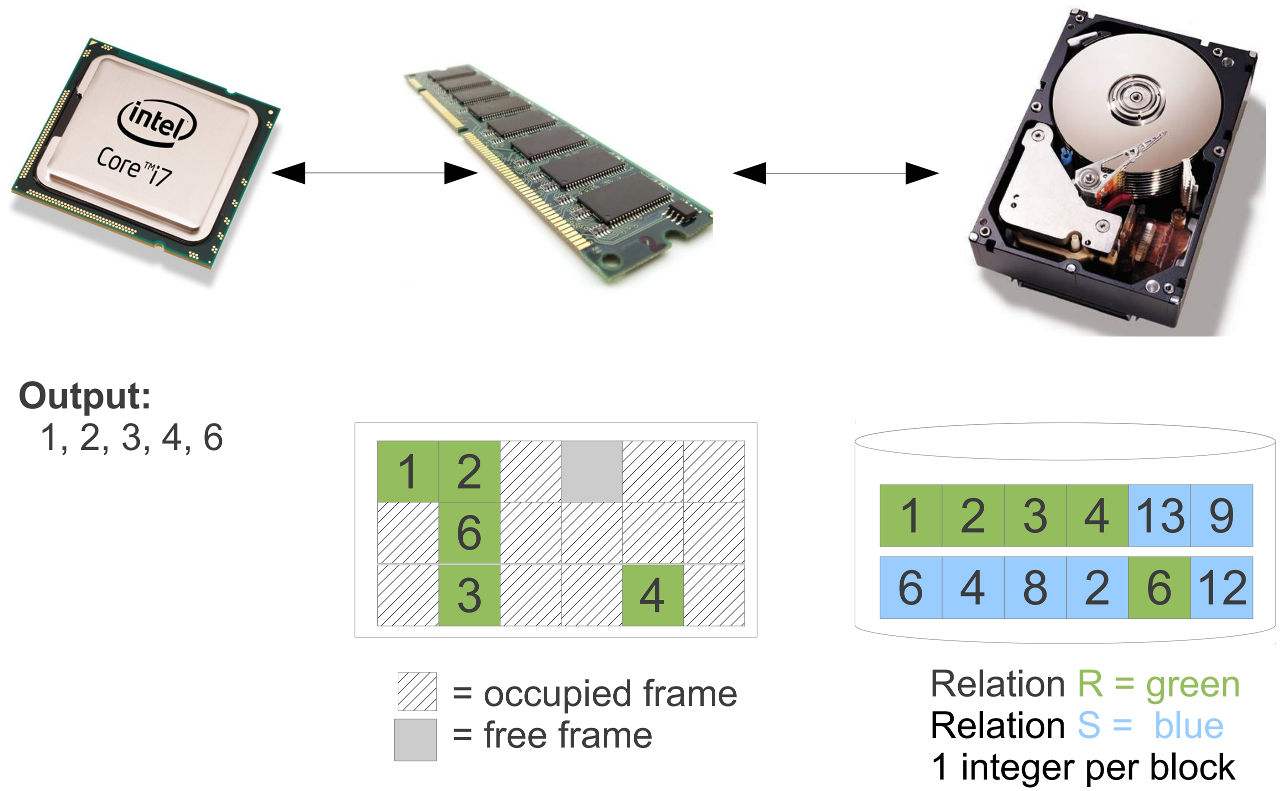


Assumption: we have $B(R) + 1$ free buffer frames

- Load all of R's blocks into memory (using $B(R)$ buffer frames) and output their elements.

Physical Operators

One-pass set union $R \cup_S S$

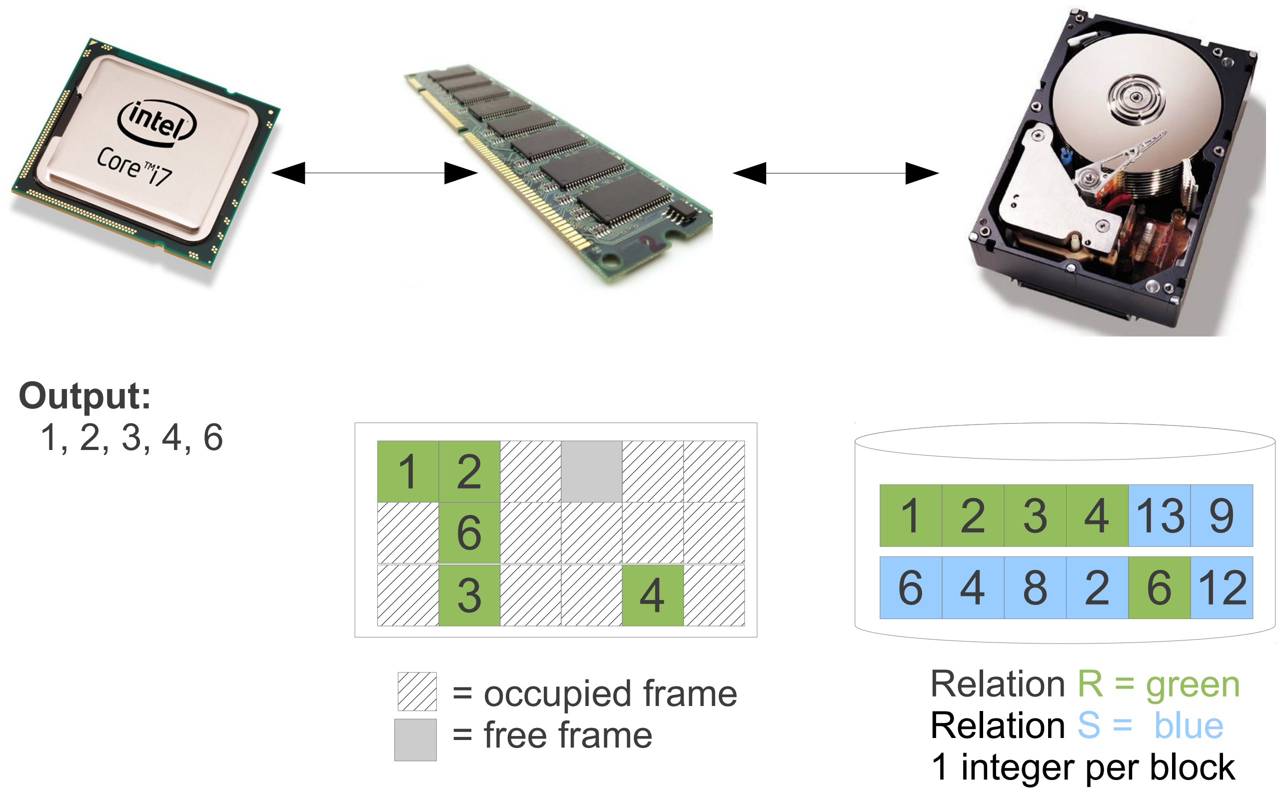


Assumption: we have $B(R) + 1$ free buffer frames

- Load all of R's blocks into memory (using $B(R)$ buffer frames) and output their elements.

Physical Operators

One-pass set union $R \cup_S S$

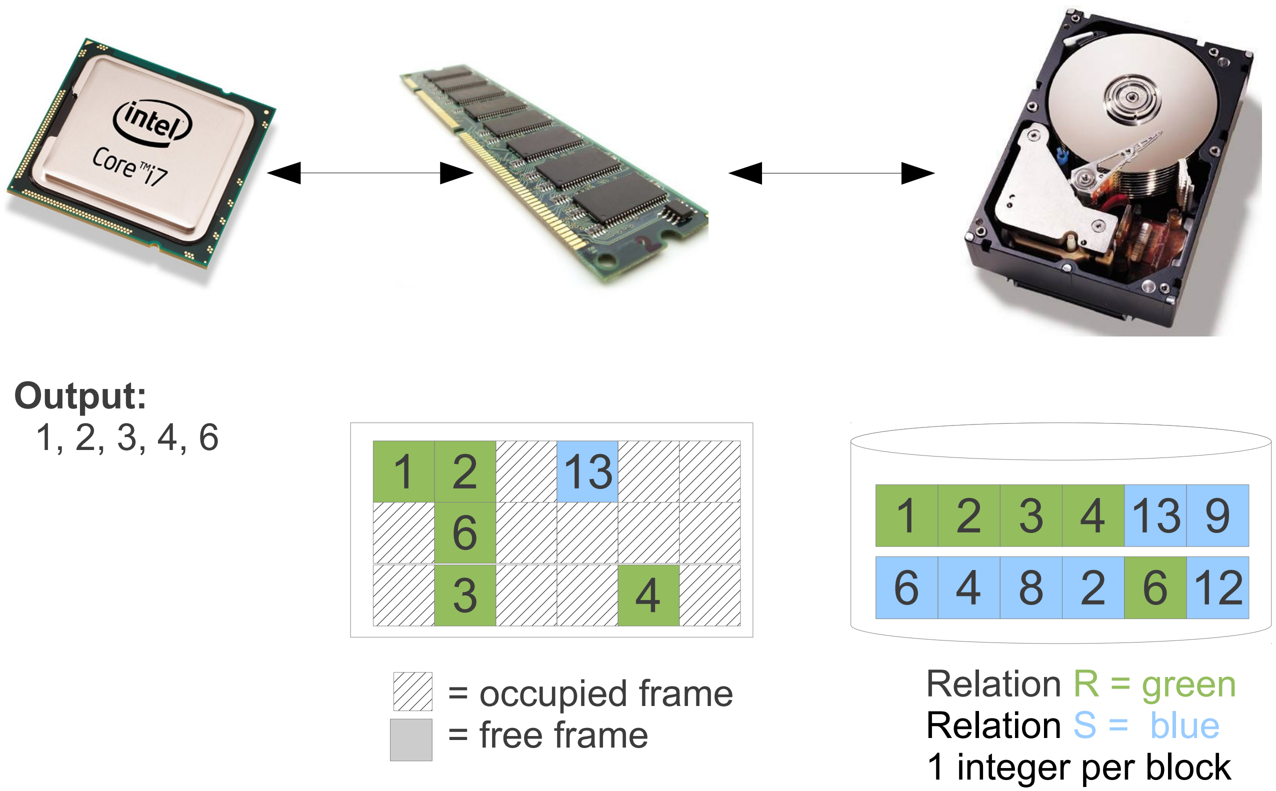


Assumption: we have $B(R) + 1$ free buffer frames

- Load 1st block of S (using 1 buffer frame), and output all of its elements that do not occur in the frames containing R.

Physical Operators

One-pass set union $R \cup_S S$

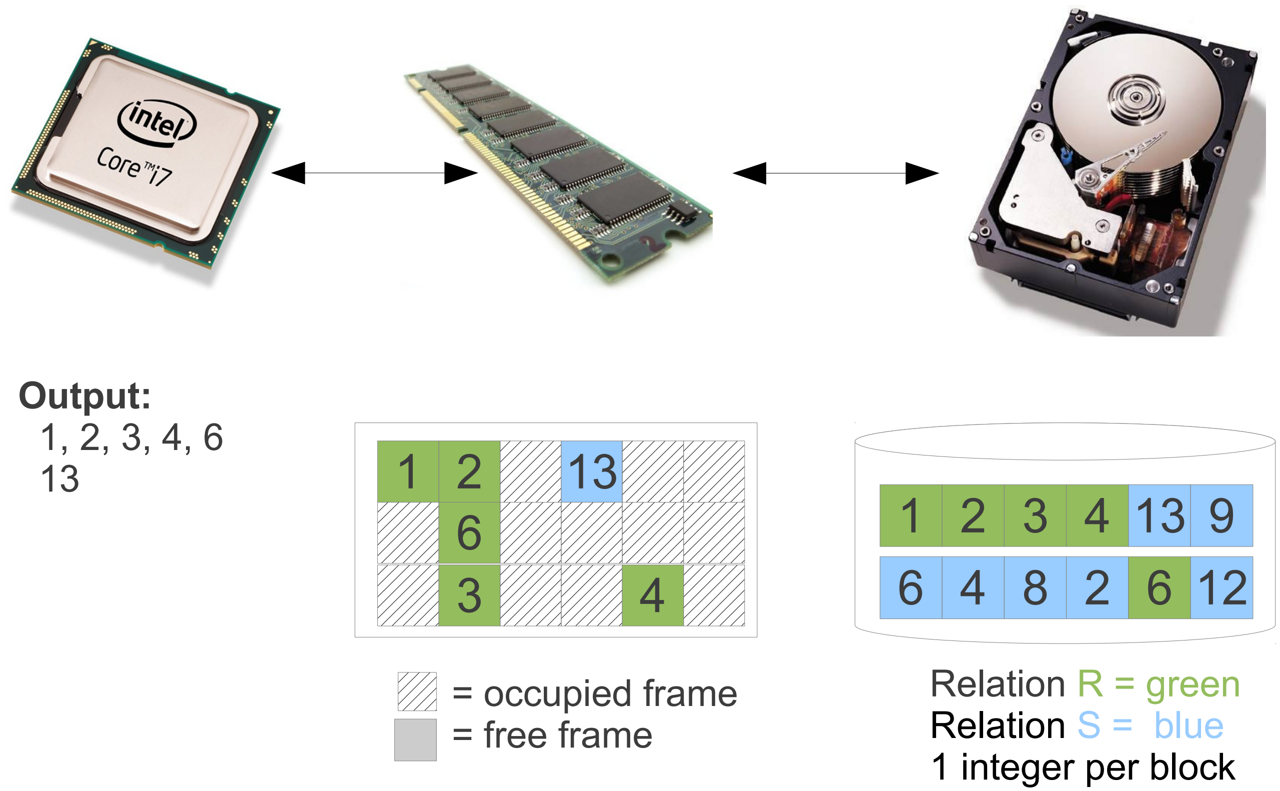


Assumption: we have $B(R) + 1$ free buffer frames

- Load 1st block of S (using 1 buffer frame), and output all of its elements that do not occur in the frames containing R.

Physical Operators

One-pass set union $R \cup_S S$

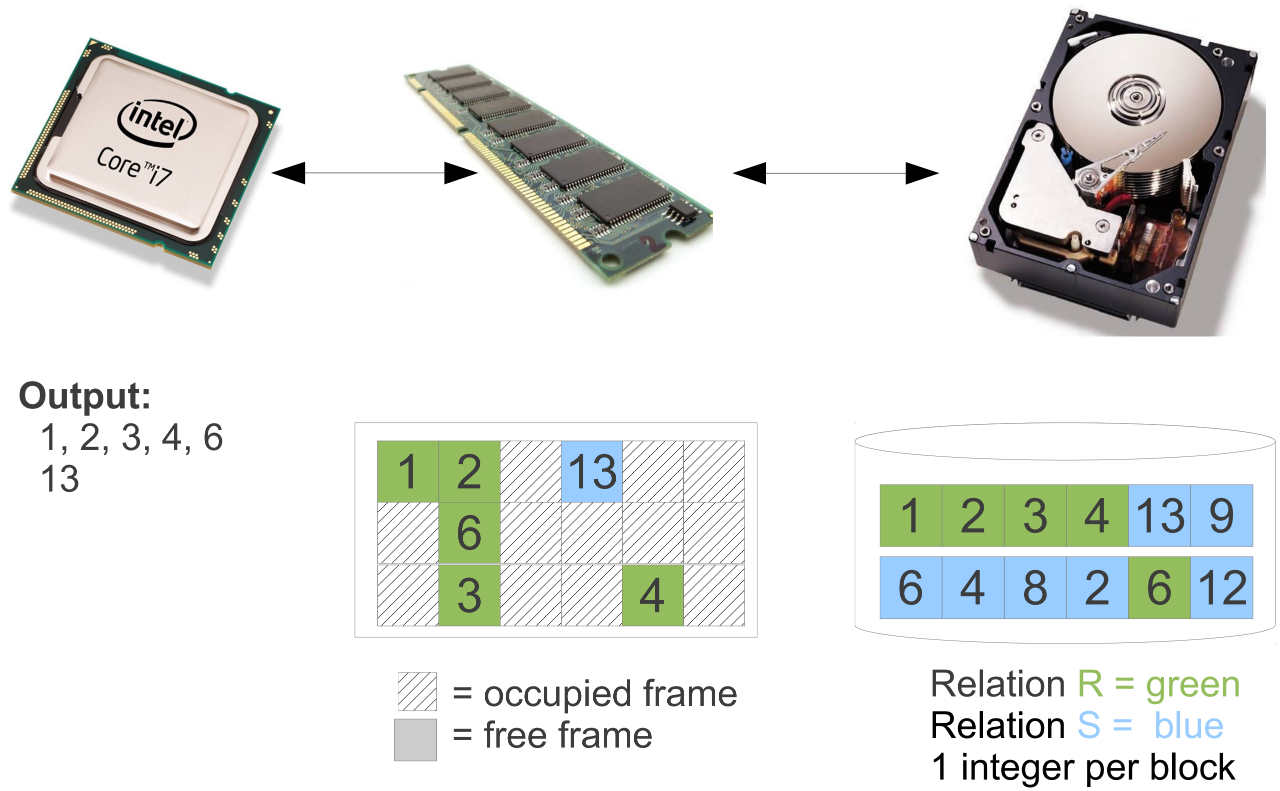


Assumption: we have $B(R) + 1$ free buffer frames

- Load 1st block of S (using 1 buffer frame), and output all of its elements that do not occur in the frames containing R.

Physical Operators

One-pass set union $R \cup_S S$

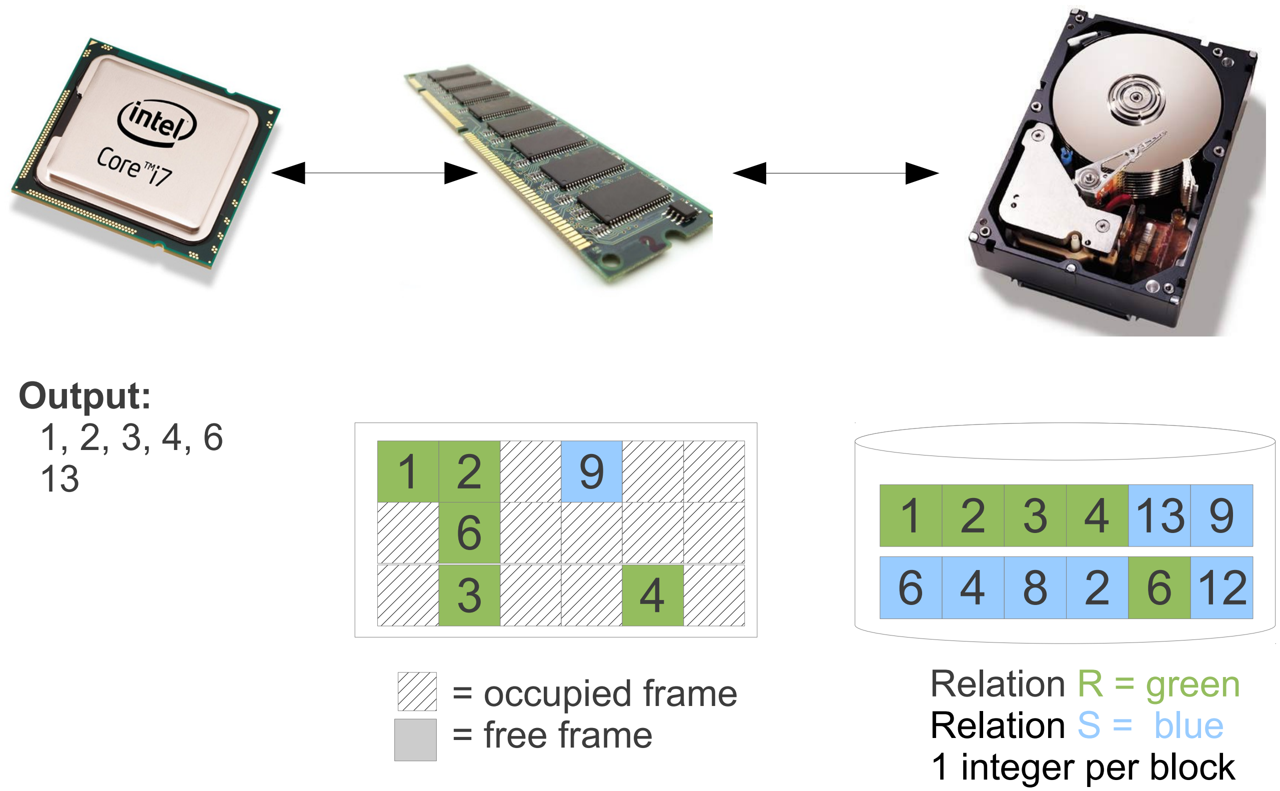


Assumption: we have $B(R) + 1$ free buffer frames

- Load 2nd block of S (using 1 buffer frame), and output all of its elements that do not occur in the frames containing R.

Physical Operators

One-pass set union $R \cup_S S$

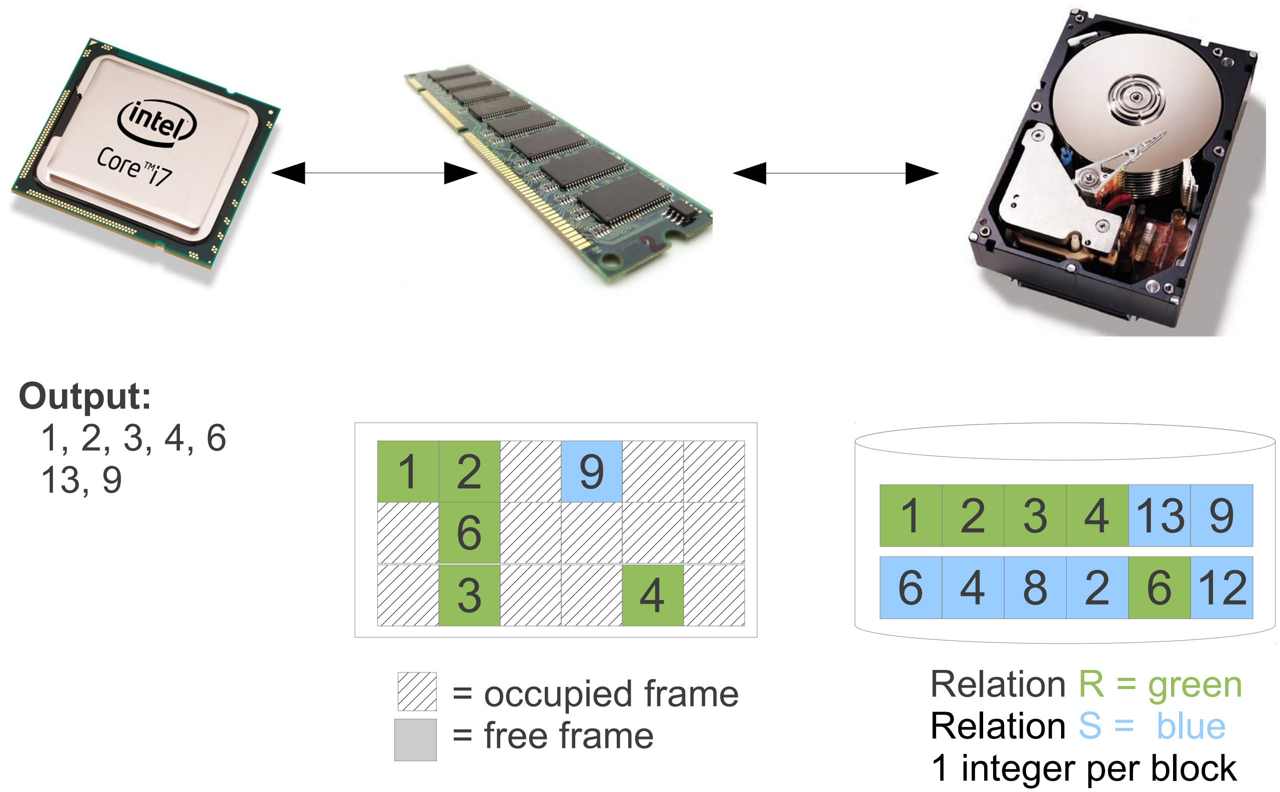


Assumption: we have $B(R) + 1$ free buffer frames

- Load 2nd block of S (using 1 buffer frame), and output all of its elements that do not occur in the frames containing R.

Physical Operators

One-pass set union $R \cup_S S$

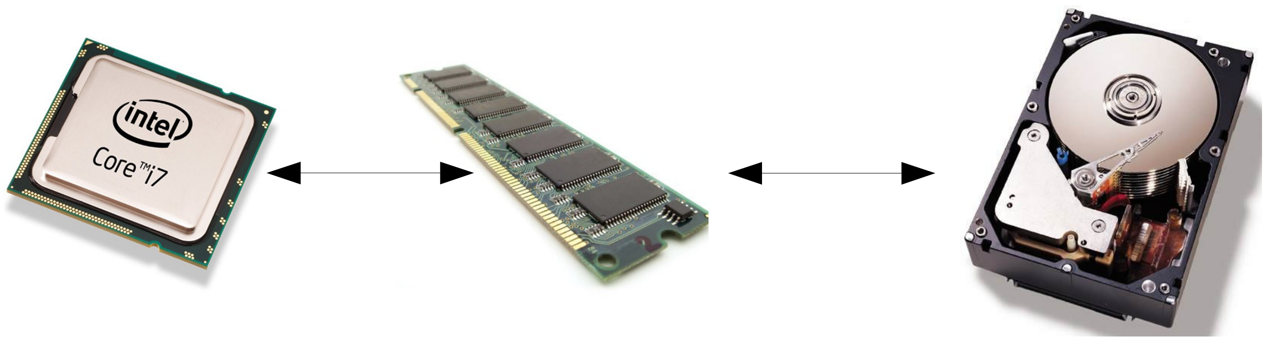


Assumption: we have $B(R) + 1$ free buffer frames

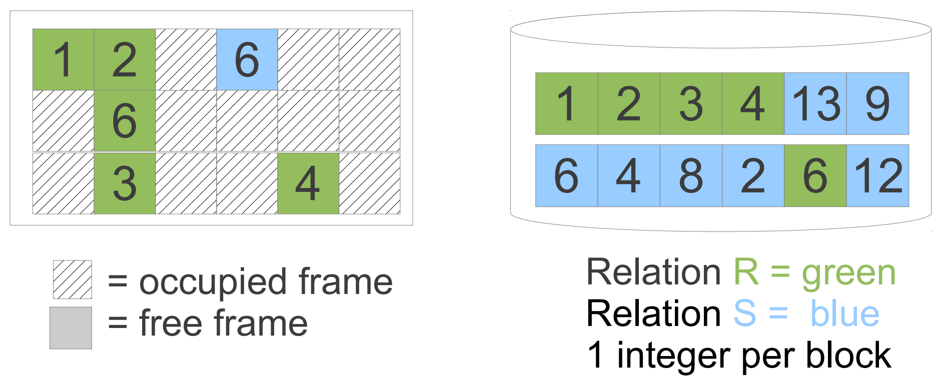
- Load 2nd block of S (using 1 buffer frame), and output all of its elements that do not occur in the frames containing R.

Physical Operators

One-pass set union $R \cup_S S$



Output:
1, 2, 3, 4, 6
13, 9

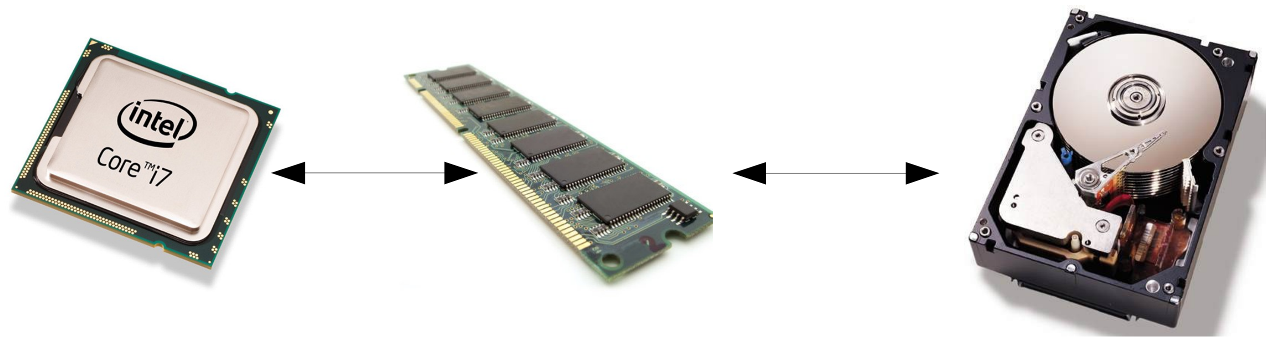


Assumption: we have $B(R) + 1$ free buffer frames

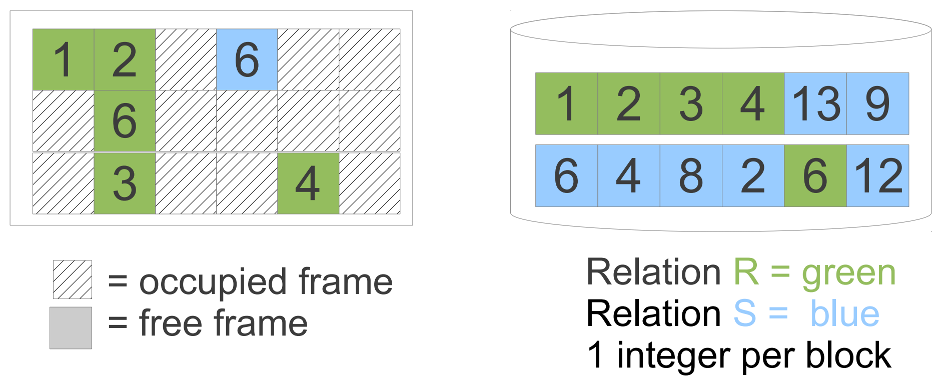
- Load 3rd block of S (using 1 buffer frame), and output all of its elements that do not occur in the frames containing R.

Physical Operators

One-pass set union $R \cup_S S$



Output:
1, 2, 3, 4, 6
13, 9

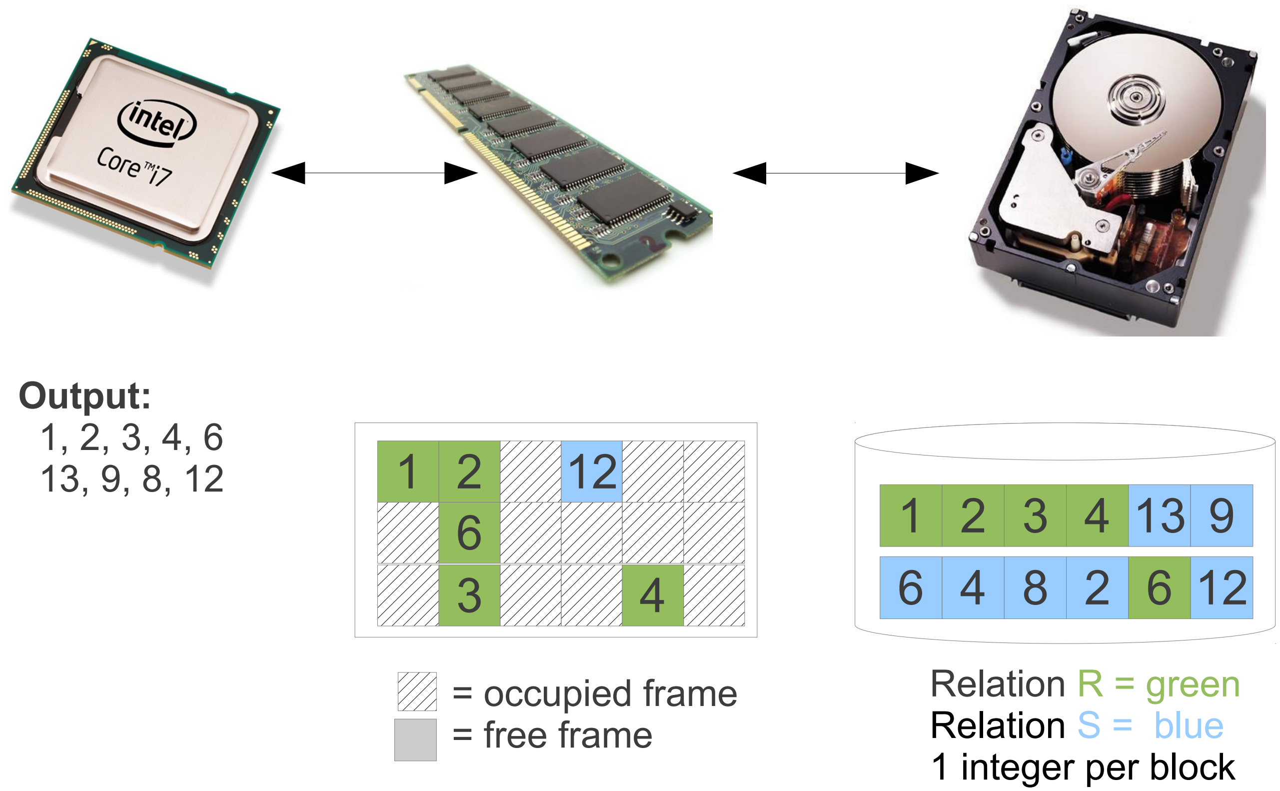


Assumption: we have $B(R) + 1$ free buffer frames

- ...and continue doing this for until the end of S is reached.

Physical Operators

One-pass set union $R \cup_S S$



Assumption: we have $B(R) + 1$ free buffer frames

- . . . and continue doing this for until the end of S is reached.

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

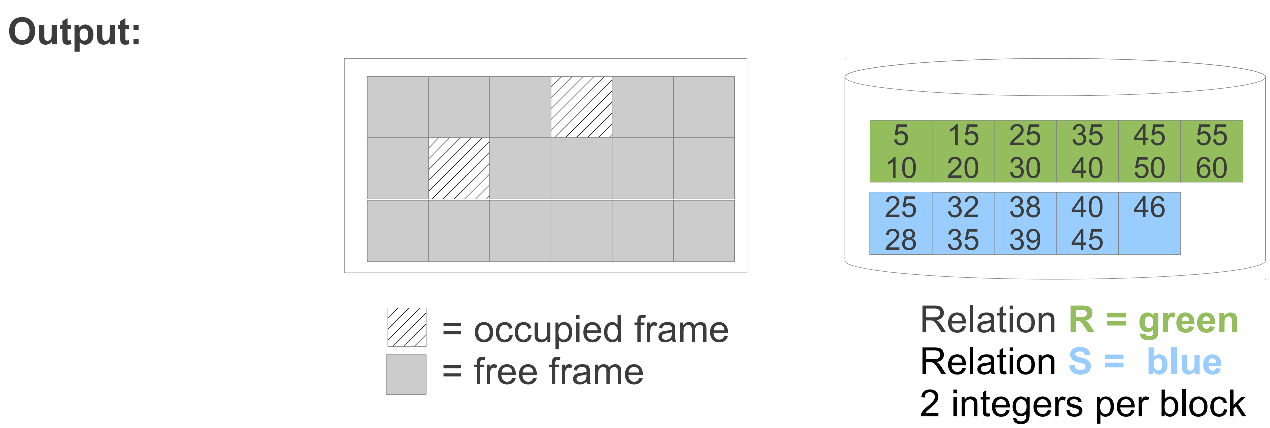
- Cost: $B(R) + B(S)$ I/O operations (ignoring output-cost)
- Note that it also costs time to check whether t_S occurs in $N_1, \ldots, N_{B(R)}$. By using a suitable main-memory data structure this can be done in $O(n)$ or $O(n \log n)$ time. We ignore this cost.
- Requires $B(R) \leq M - 1$

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.

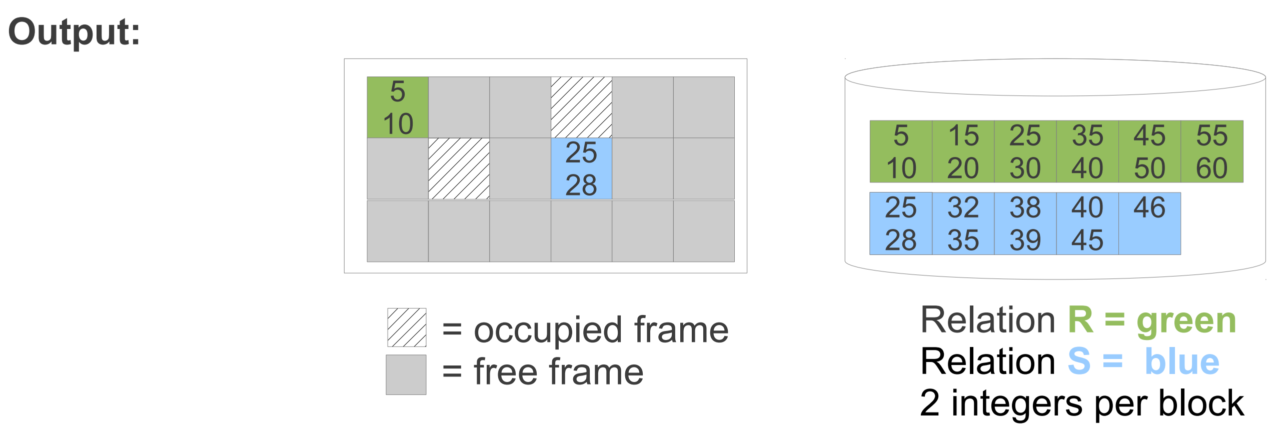


Physical Operators

Sort-based set union

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.

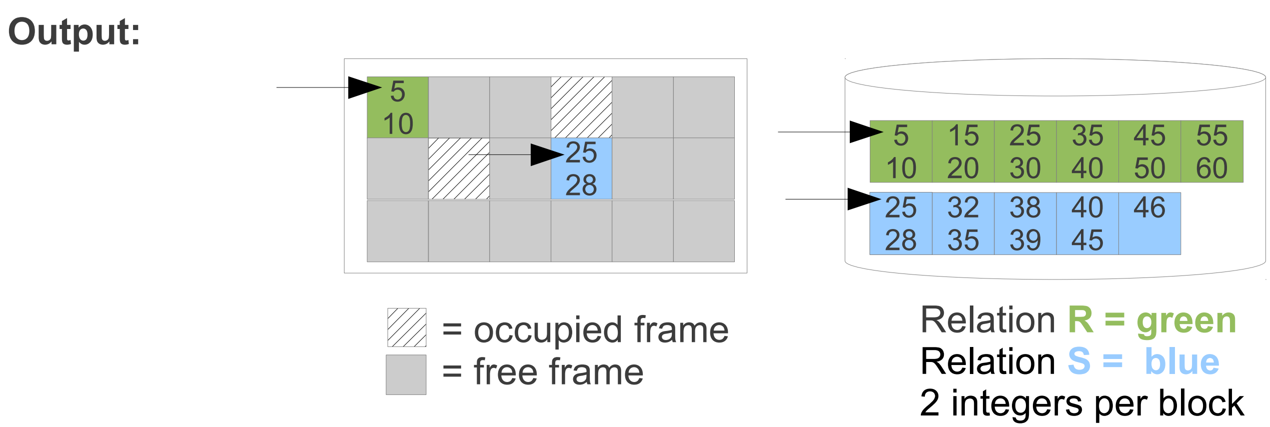


Physical Operators

Sort-based set union

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.

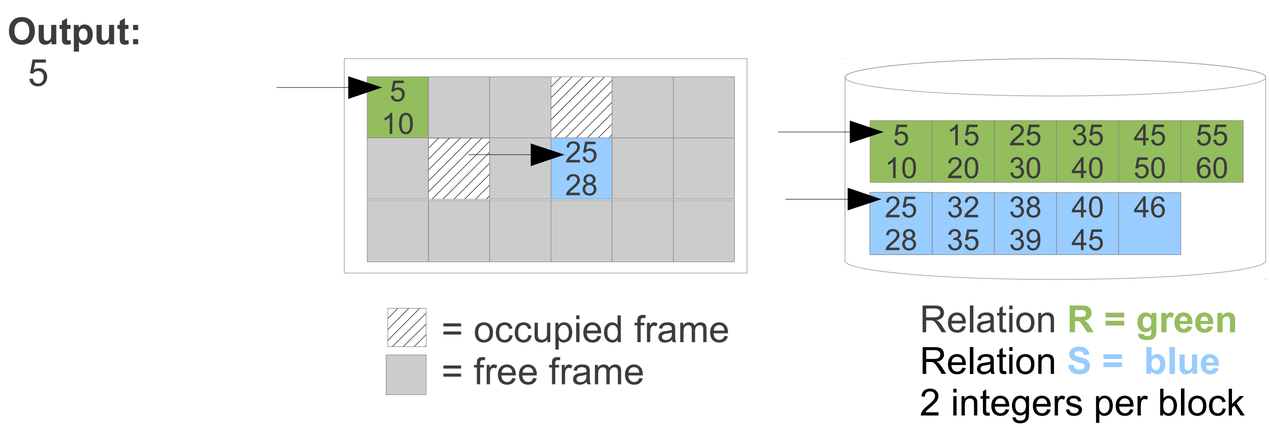


Physical Operators

Sort-based set union

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.

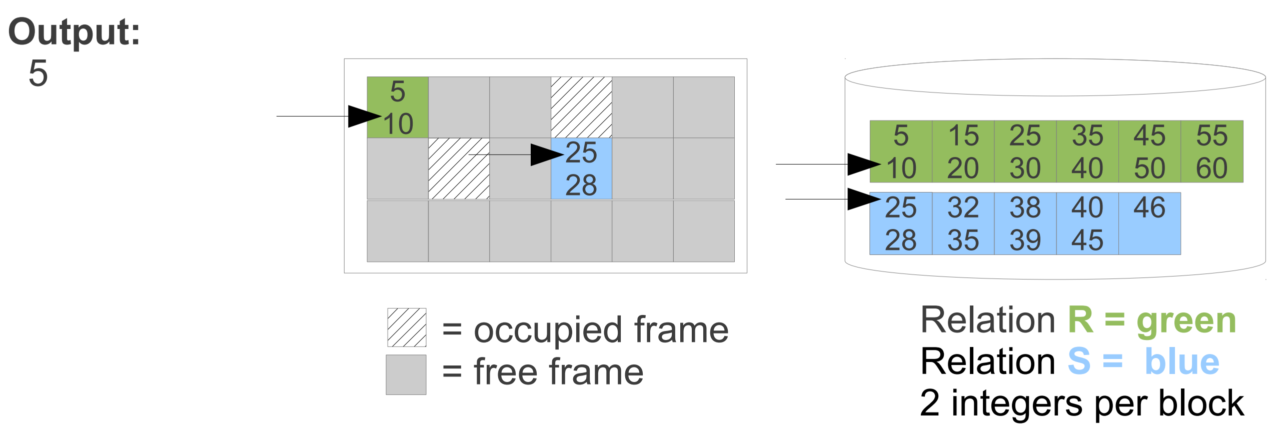


Physical Operators

Sort-based set union

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.

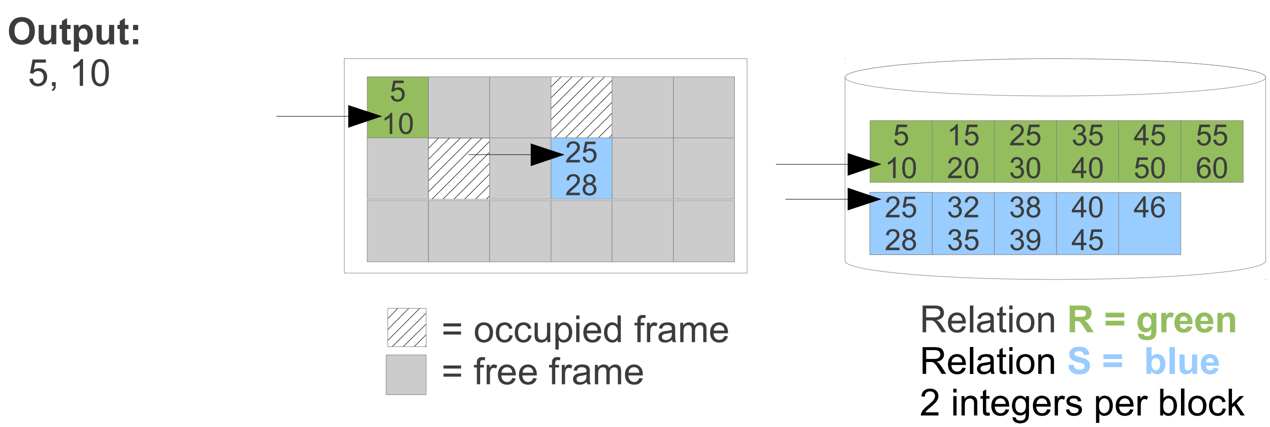


Physical Operators

Sort-based set union

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.

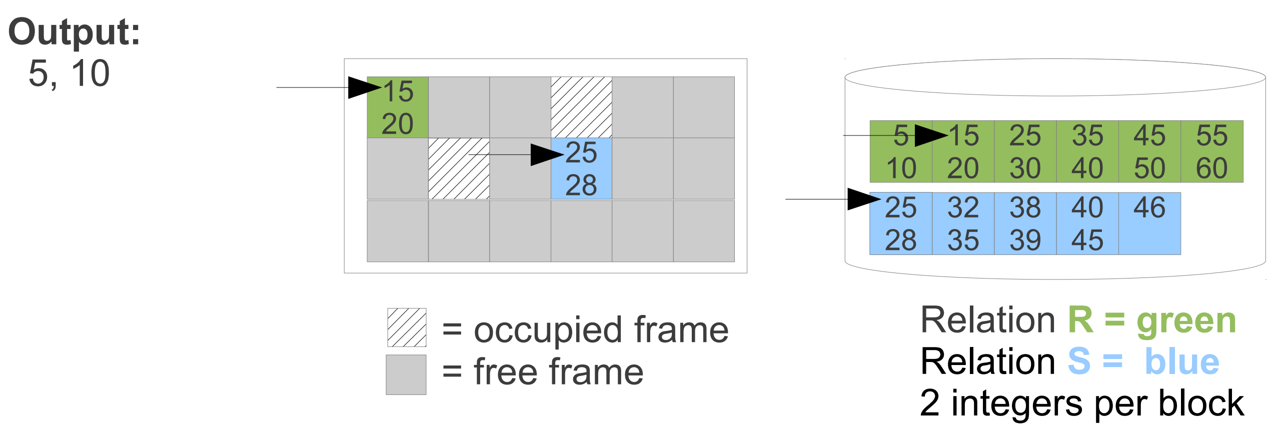


Physical Operators

Sort-based set union

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.

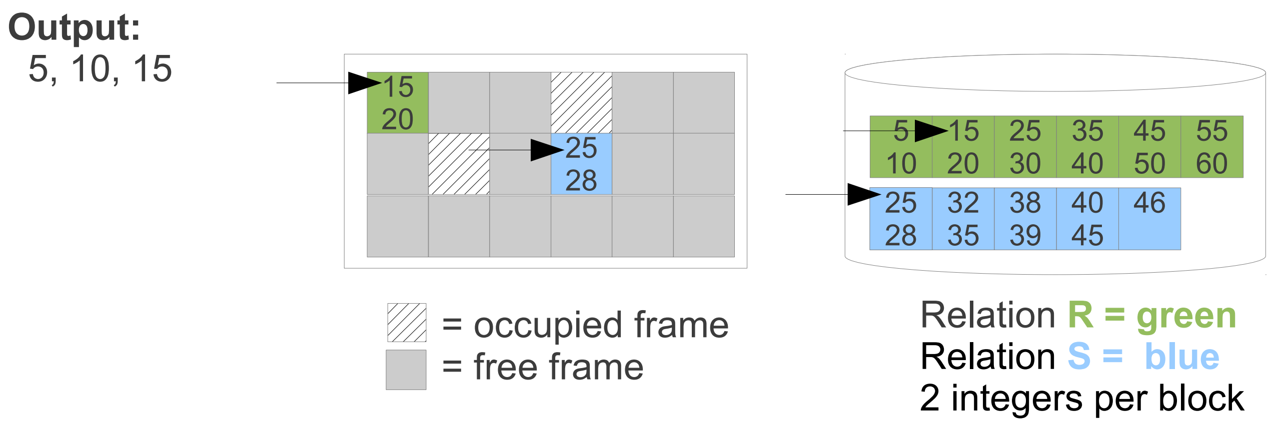


Physical Operators

Sort-based set union

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.

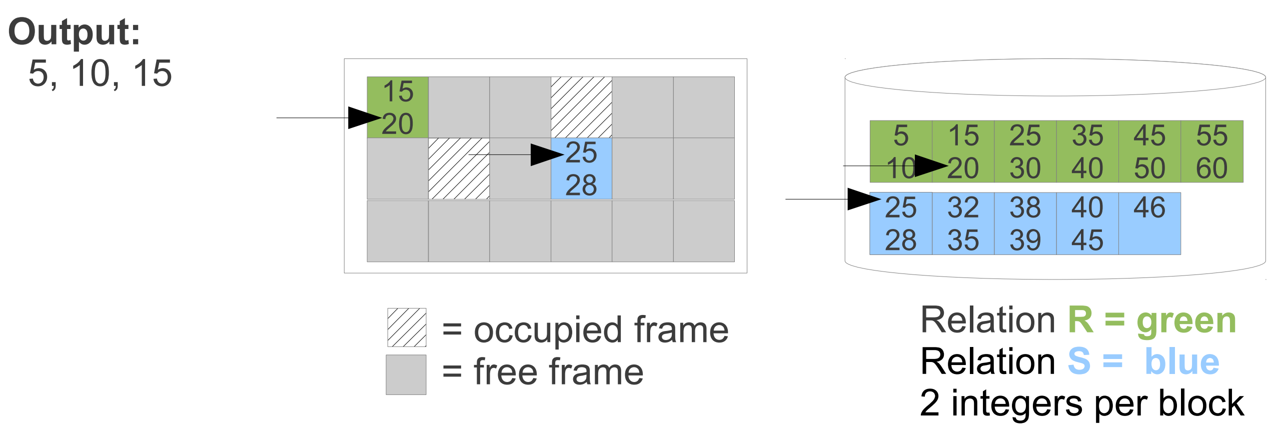


Physical Operators

Sort-based set union

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.

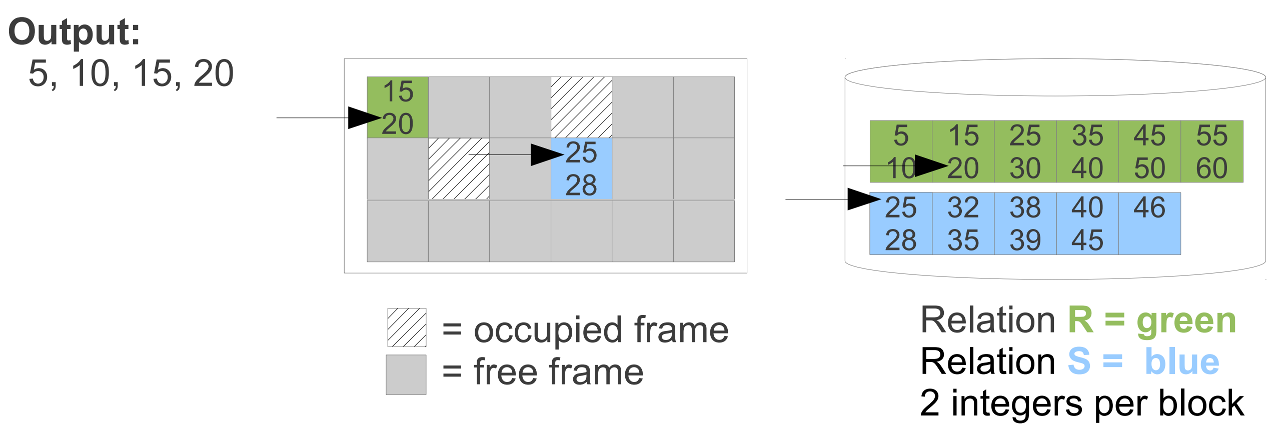


Physical Operators

Sort-based set union

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.

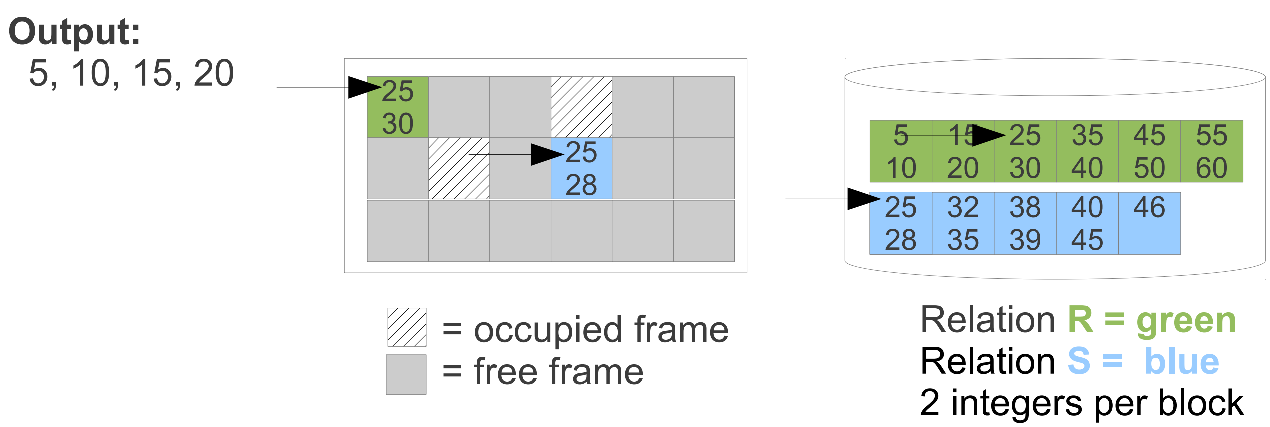


Physical Operators

Sort-based set union

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.

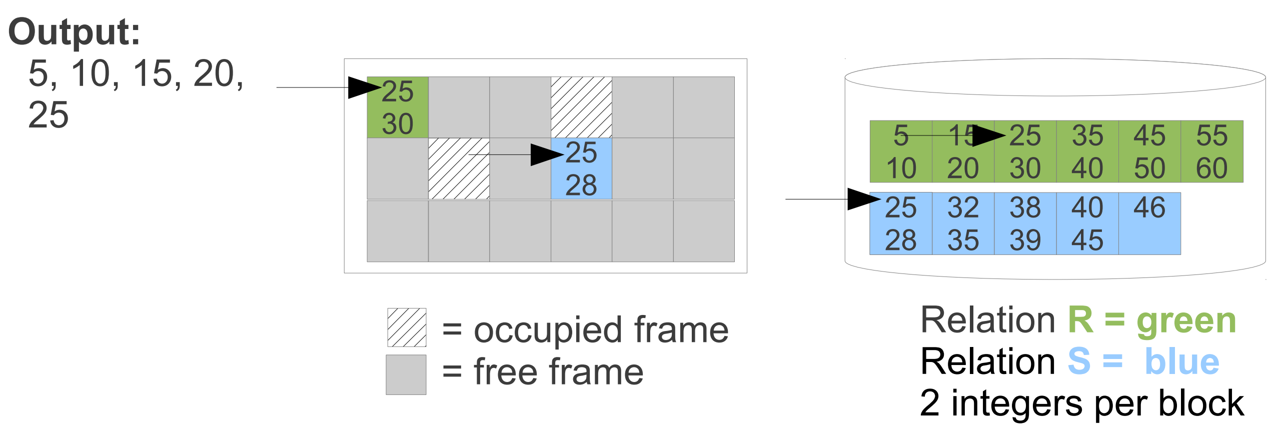


Physical Operators

Sort-based set union

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.

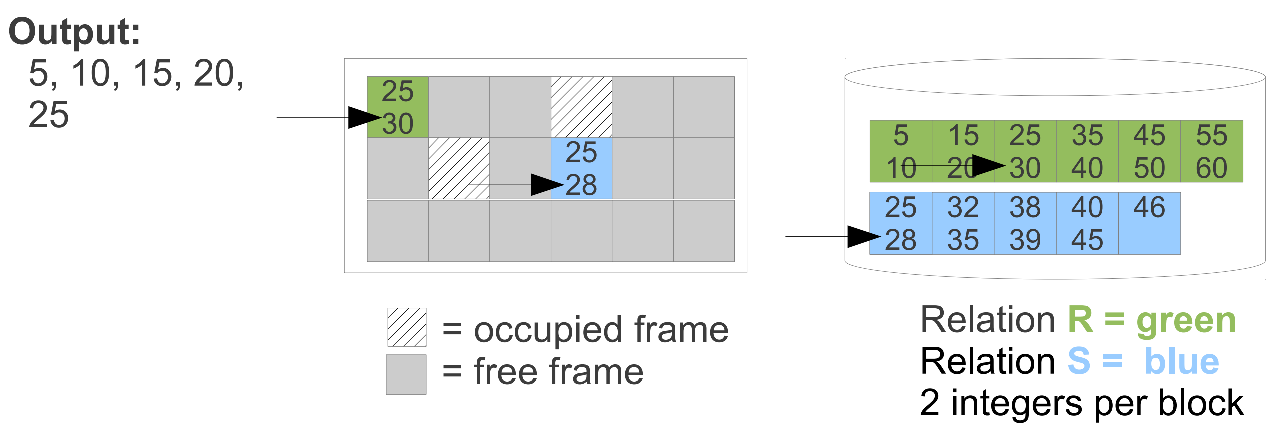


Physical Operators

Sort-based set union

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.

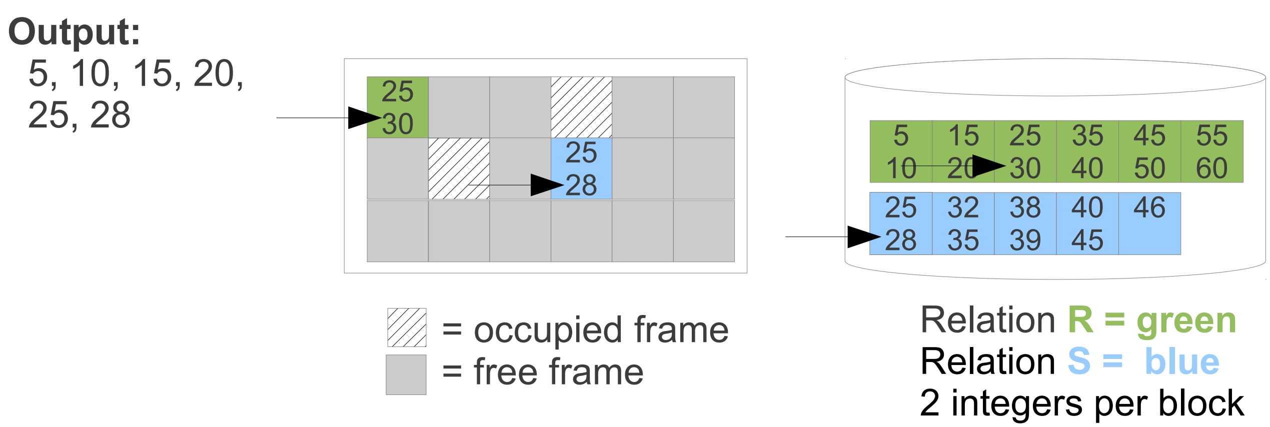


Physical Operators

Sort-based set union

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.

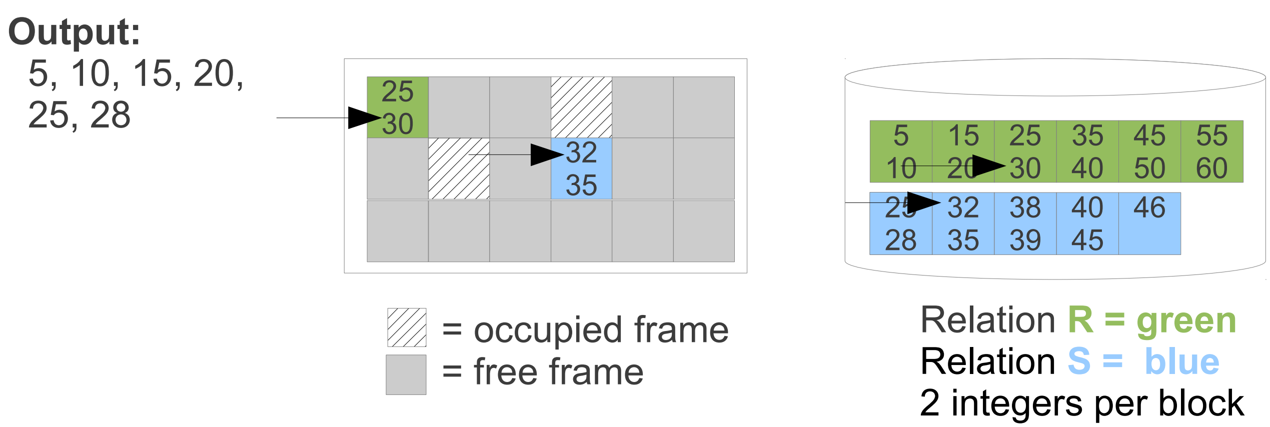


Physical Operators

Sort-based set union

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.

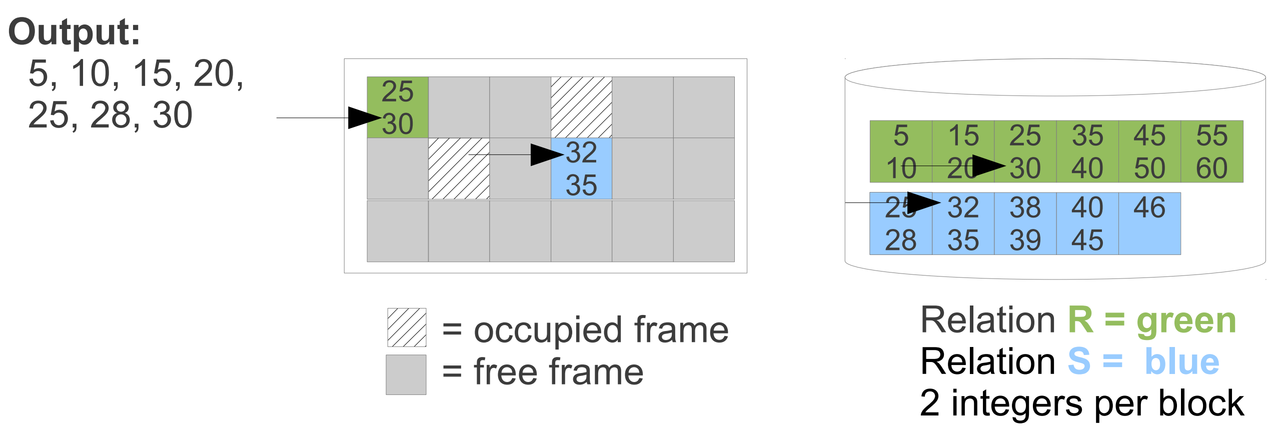


Physical Operators

Sort-based set union

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

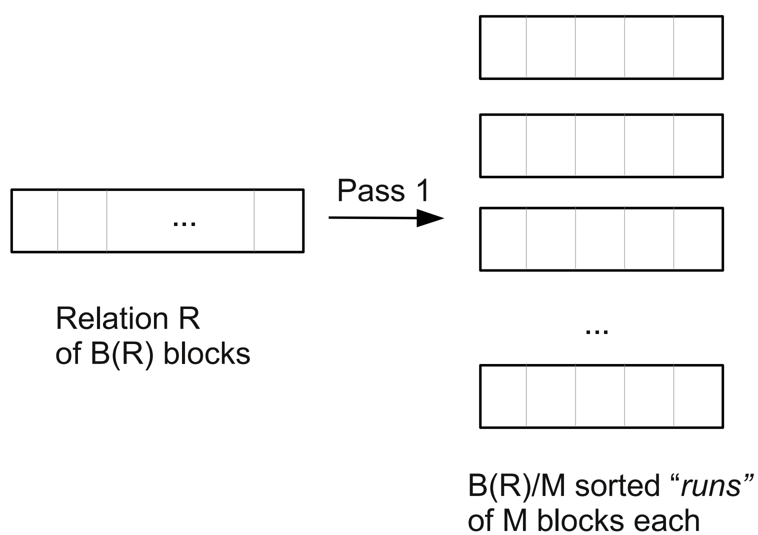
We can also alternatively compute the set union $R \cup_S S$ as follows (again R and S are assumed to be sets):

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. Assume that we are currently at tuple t_R in R and tuple t_S in S:
   - If $t_R < t_S$ then we output t_R and move t_R to the next tuple in R (possibly by loading the next block of R into memory).
   - If $t_R > t_S$ then we output t_S and move t_S to the next tuple in S (possibly by loading the next block of S into memory).
   - If $t_R = t_S$ then we output t_R and move t_R to the next tuple in R and t_S to the next tuple in S (possibly by loading the next block)

Physical Operators

Sort-based set union

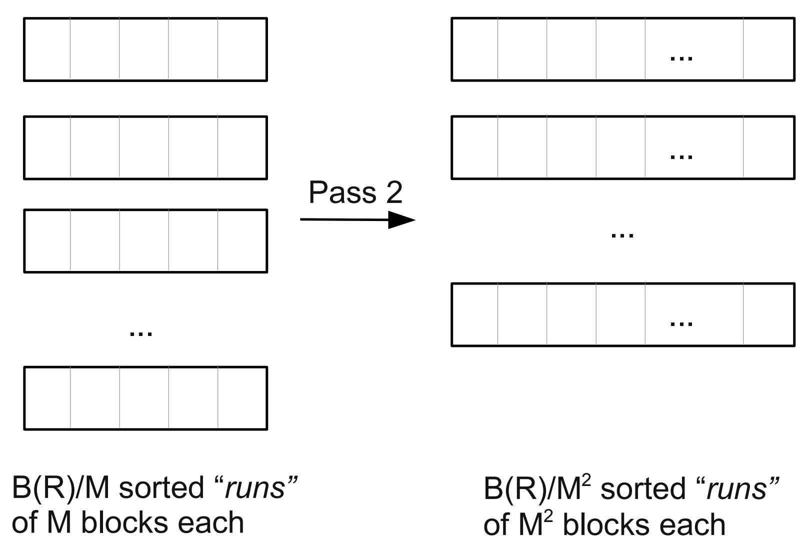
- Sorting can in principle be done by any suitable algorithm, but is usually done by Multiway Merge-Sort:
  - In the first pass we read M blocks at the same time from the input relation, sort these by means of a main-memory sorting algorithm, and write the sorted resulting sublist to disk. After the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



Physical Operators

Sort-based set union

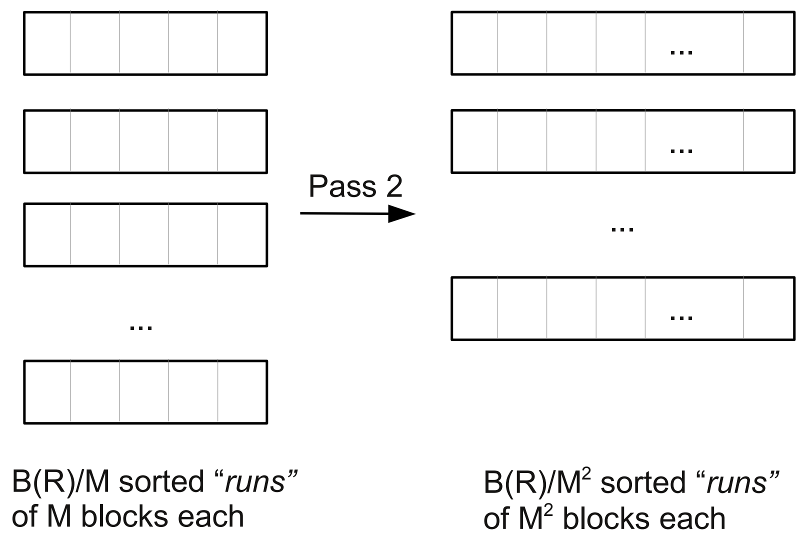
- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  - In the 2nd pass, we merge the first M sublists from the first pass into a single sublist of M^2 blocks. We do so by iterating synchronously over these M sublists, keeping 1 block of each list into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



Physical Operators

Sort-based set union

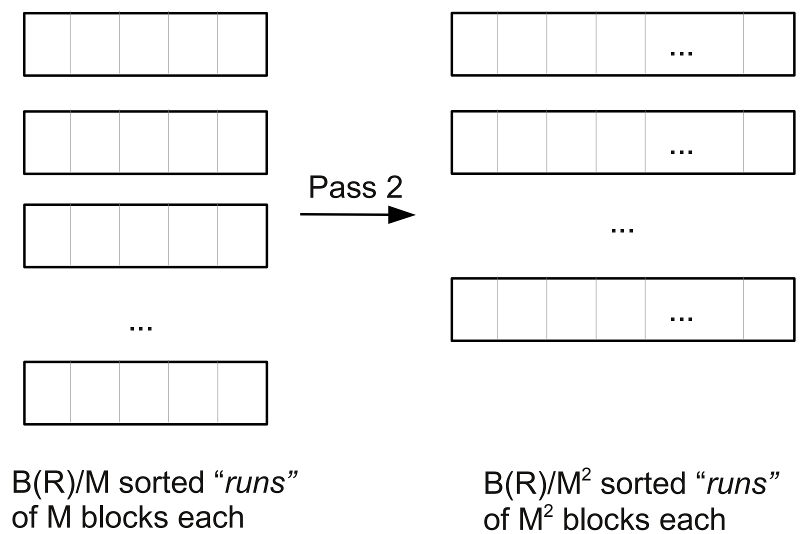
- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  - We then merge the next M sublists into a single sublist, and continue until we have treated each sublist resulting from the first pass.



Physical Operators

Sort-based set union

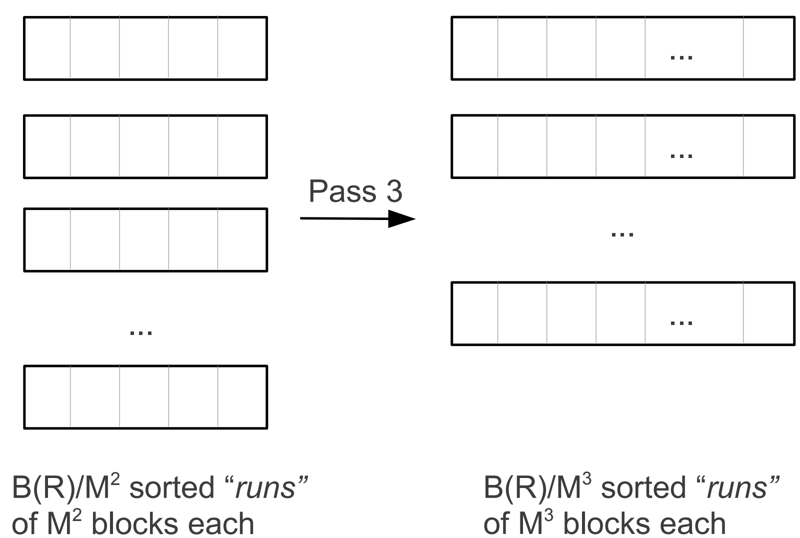
- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  - After the second pass we hence have $B(R)/M^2$ sorted sublists of M^2 blocks each.



Physical Operators

Sort-based set union

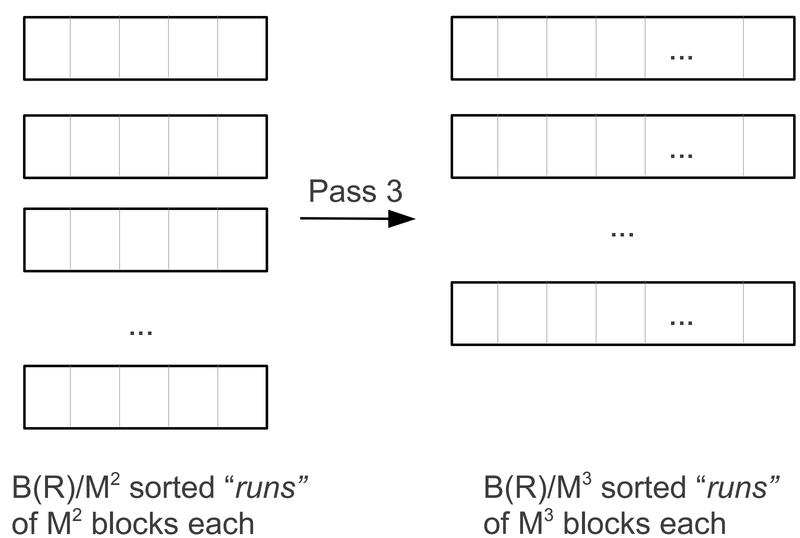
- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  - In the 3rd pass, we merge the first M sublists from the 2nd pass (each of M^2 blocks) into a single sublist of M^3 blocks. We do so by iterating synchronously over these M sublists, keeping 1 block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



Physical Operators

Sort-based set union

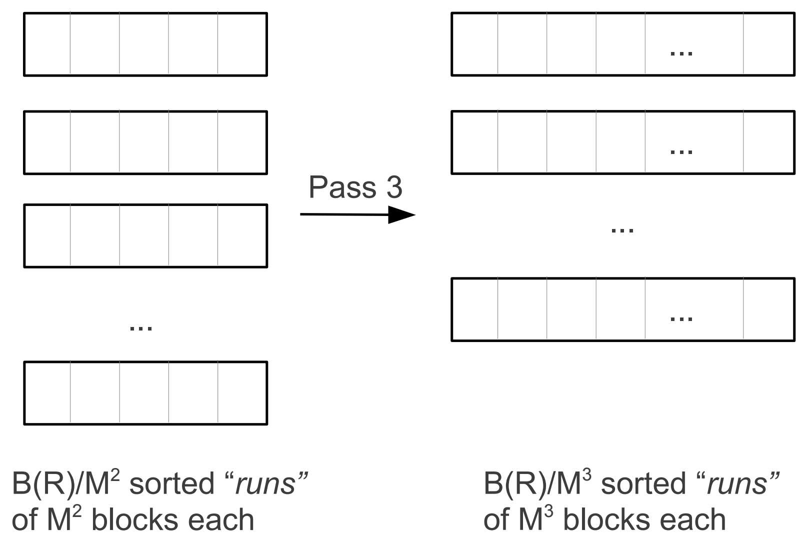
- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  - We then merge the next M sublists into a single sublist, and continue until we have treated each sublist resulting from the 2nd pass .



Physical Operators

Sort-based set union

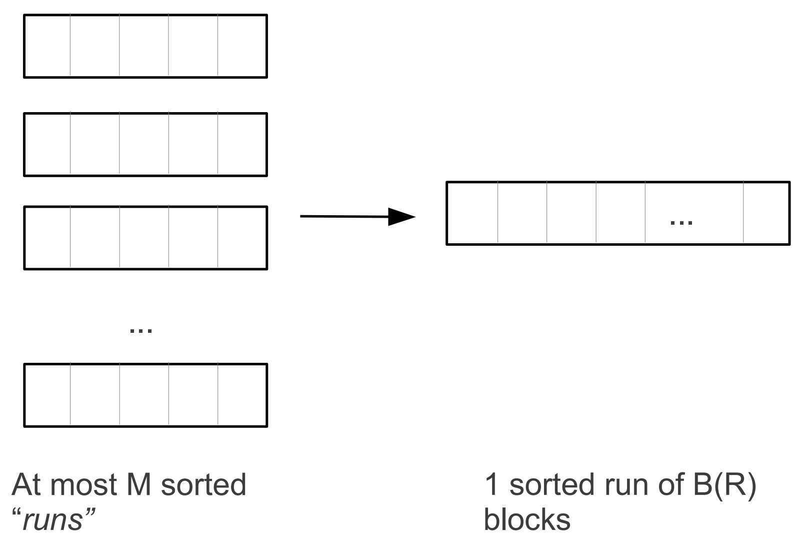
- Sorting can in principle be done by suitable algorithm, but is usually done by Multiway Merge-Sort:
  - After the 3rd pass we hence have $B(R)/M^3$ sorted sublists of M^3 blocks each.



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:

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

Sort-based set union

Example: we have 15 buffers available, $B(R) = 100$, and $B(S) = 120$.

- Number of passes required to sort R completely: $\lceil \log_M B(R) \rceil = 2$
- Number of passes required to sort S completely: $\lceil \log_M B(S) \rceil = 2$
- Can the optimization be applied?

$$\left\lceil \frac{100}{15} \right\rceil + \left\lceil \frac{120}{15} \right\rceil = 15 \leq M$$

- The optimized sort-based set union hence costs:

$$2 \times 100 \times 2 + 2 \times 120 \times 2 - 100 - 120 = 660$$

Physical Operators

Sort-based set union

- The book states that in practice 2 passes usually suffice to completely sort a relation.
- If we assume that R and S can be sorted in two passes (given the available memory M) then we can instantiate our cost formula as follows:
  - Without optimization: $5B(R) + 5B(S)$
  - With optimization: $3B(R) + 3B(S)$, but in this case we require sufficient memory:
  $$\left\lceil \frac{B(R)}{M} \right\rceil + \left\lceil \frac{B(S)}{M} \right\rceil \leq M$$
  or (approximately) $B(R) + B(S) \leq M^2$.
  $\rightarrow$ This is the formula that you will find in the book!
- Note that the book focuses on the optimized algorithm in the case where two passes suffice: the so-called "two-pass, sort-based set union". It only sketches the generalization to multiple passes.

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

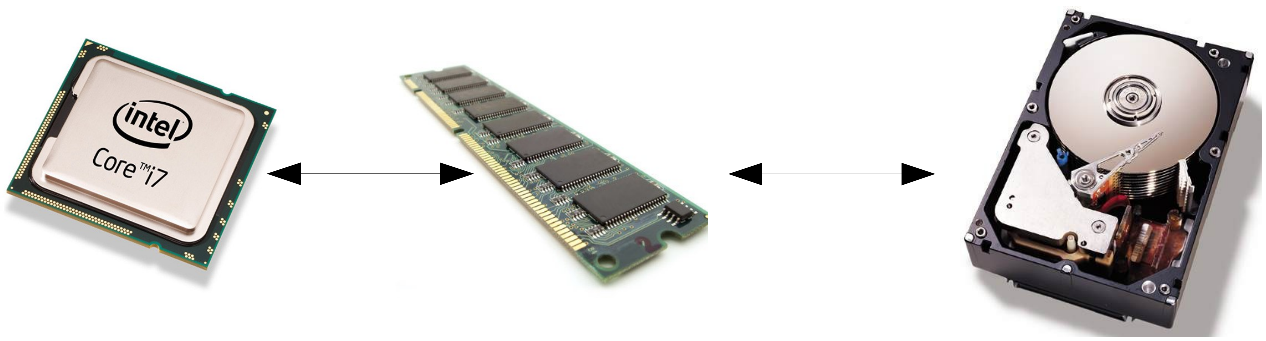
Hash-based set union

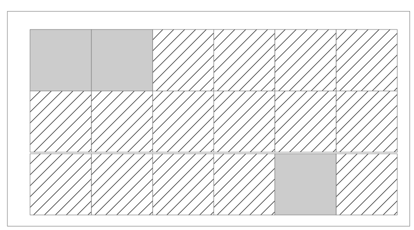
How do we partition R in buckets of at most $M - 1$ blocks?

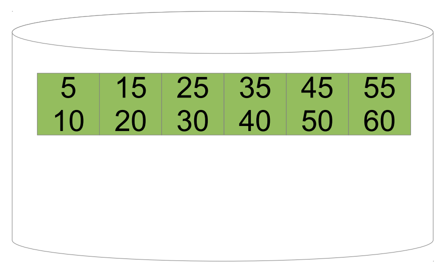
1. Using M buffers, we first hash R into $M - 1$ buckets.
2. Subsequently we partition each bucket separately in $M - 1$ new buckets, by using a new hash function distinct from the one used in the previous step (why?)
3. We continue doing so until the obtained buckets consists of at most $M - 1$ blocks.

Physical Operators

Hashing R into $M-1$ buckets using M buffers





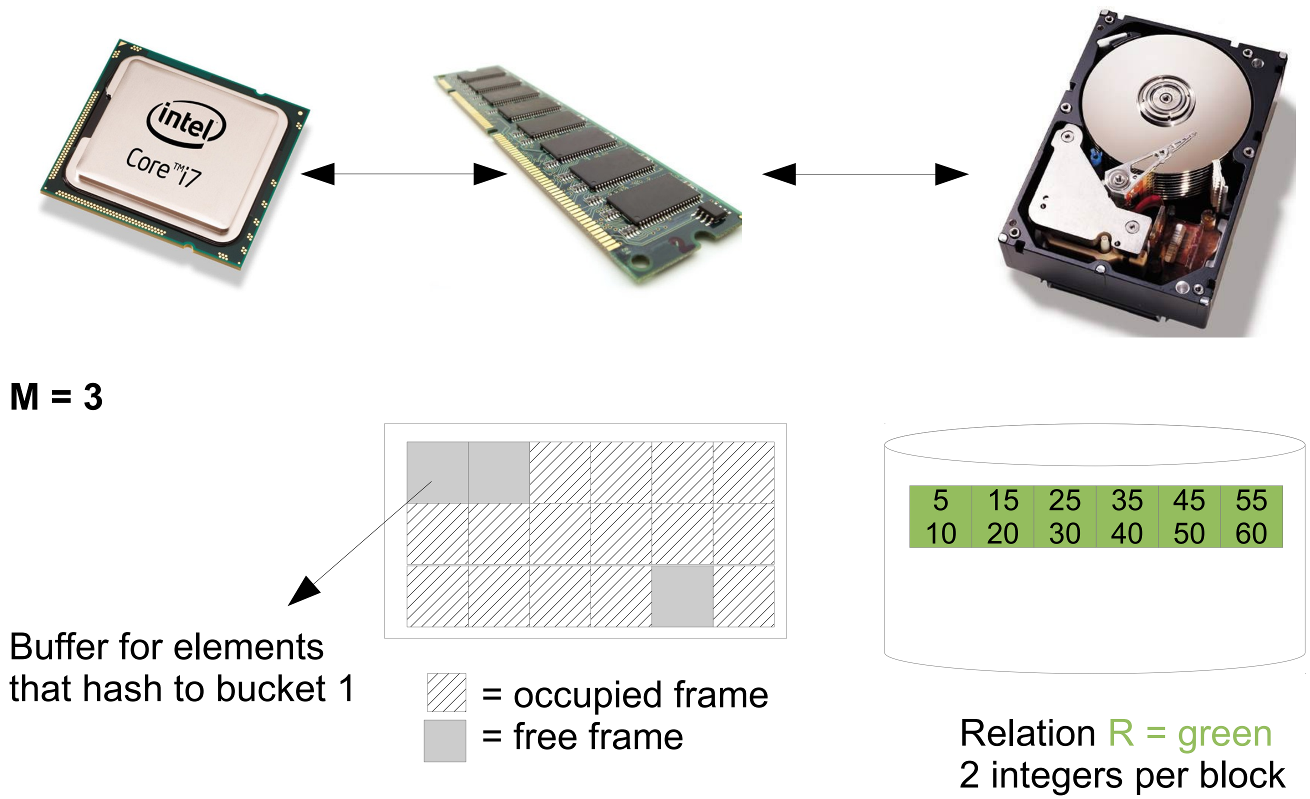


= occupied frame
= free frame

Relation R = green
2 integers per block

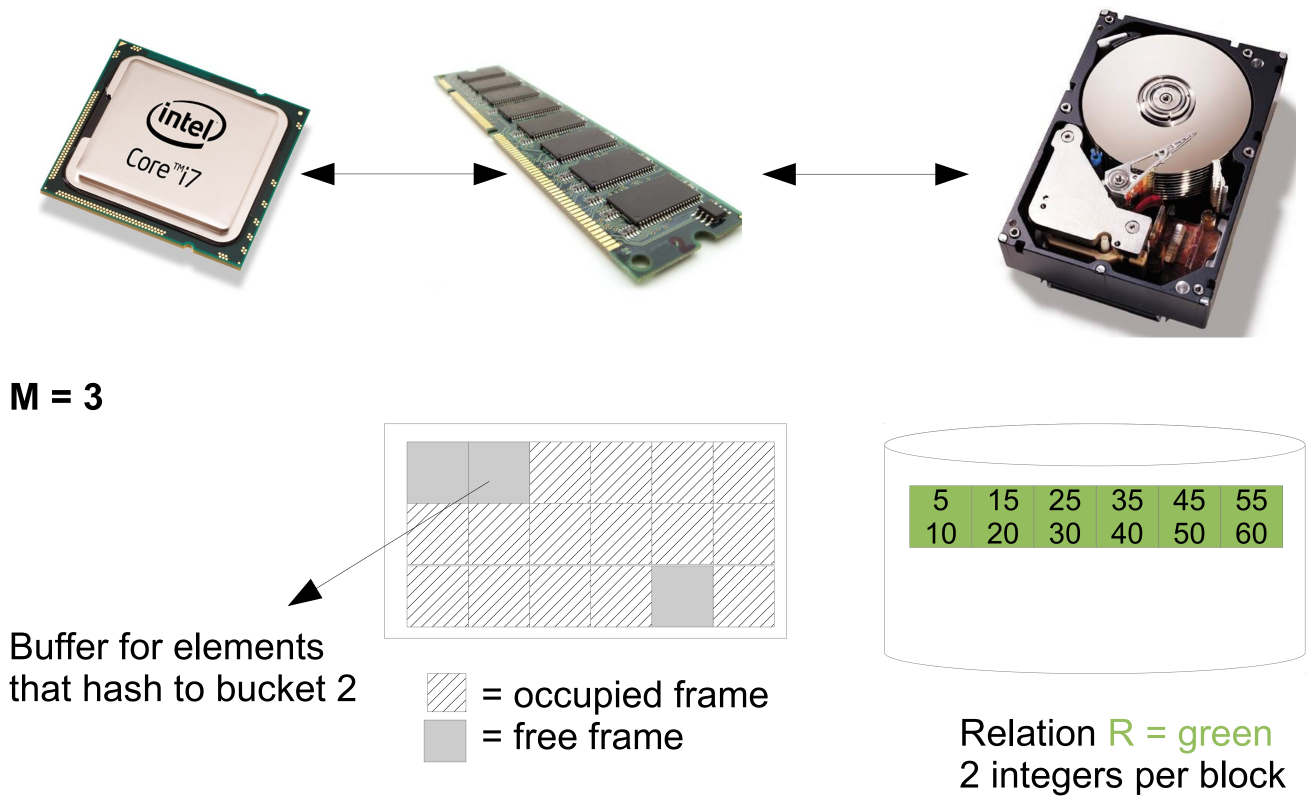
Physical Operators

Hashing R into $M-1$ buckets using M buffers



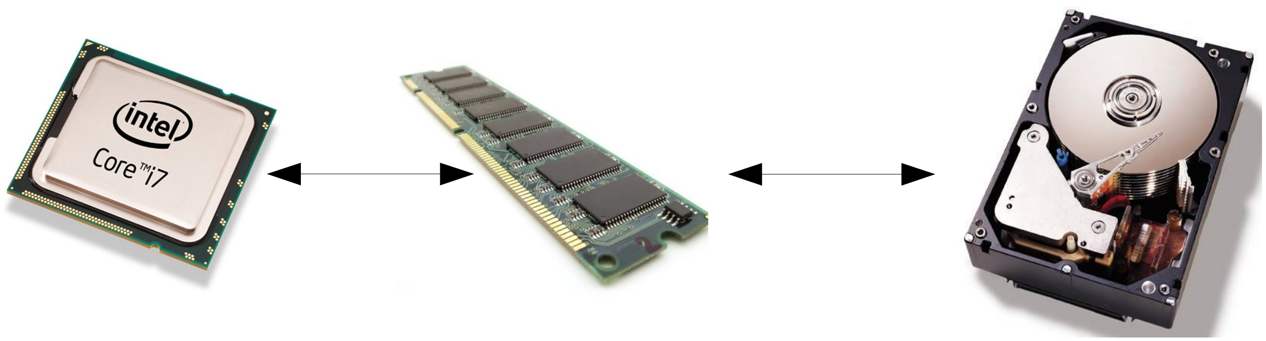
Physical Operators

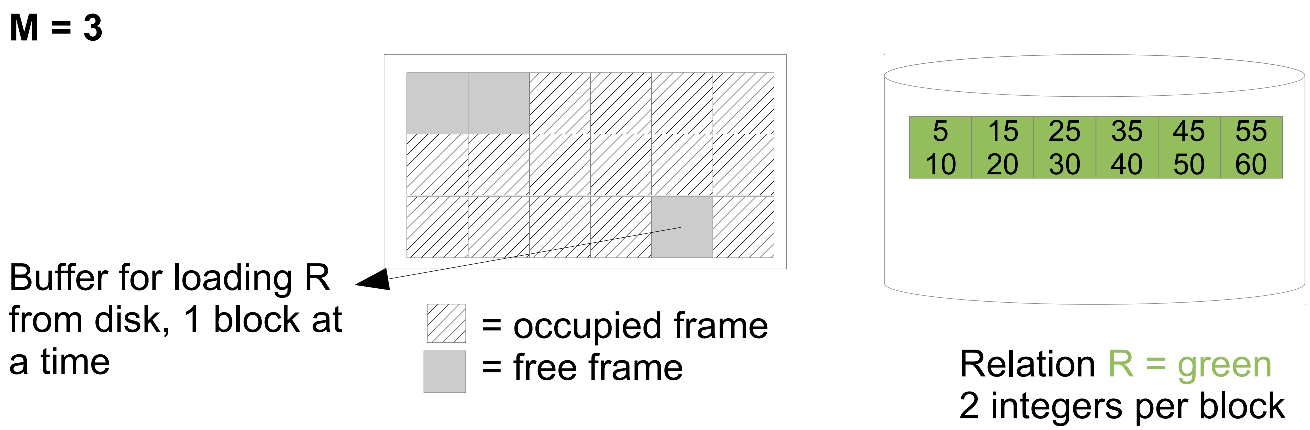
Hashing R into $M-1$ buckets using M buffers



Physical Operators

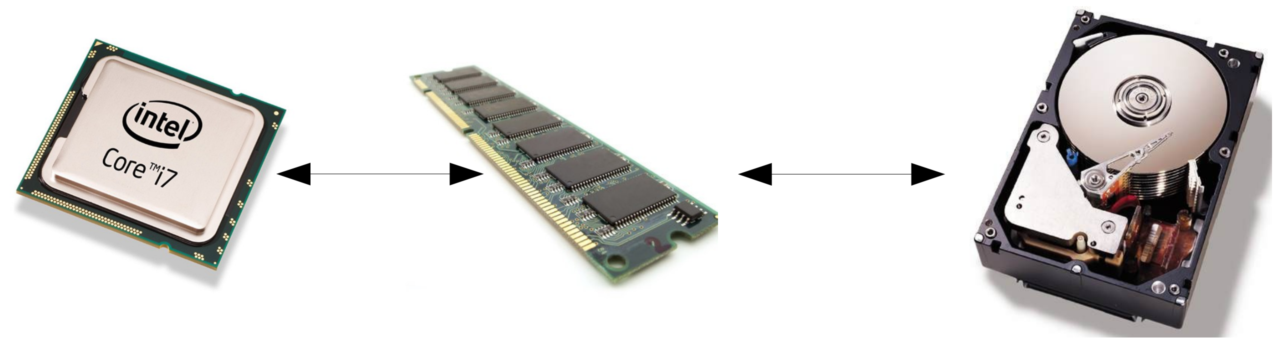
Hashing R into $M-1$ buckets using M buffers



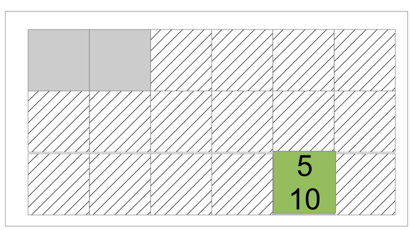


Physical Operators

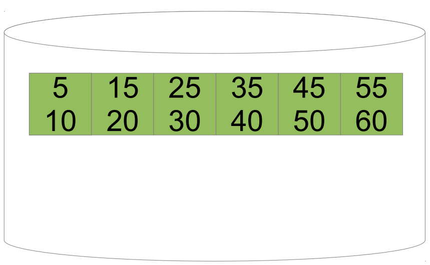
Hashing R into $M - 1$ buckets using M buffers



M = 3



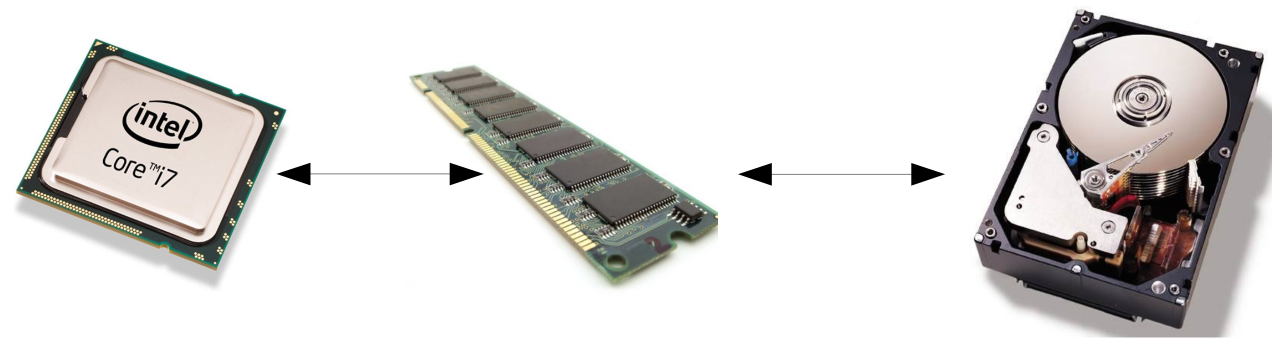
= occupied frame
= free frame



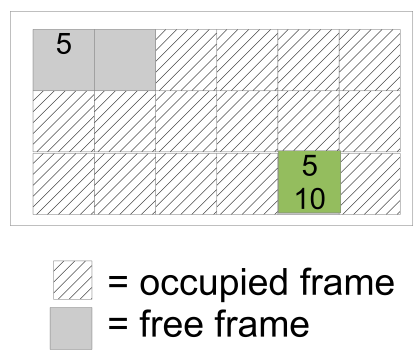
Relation R = green
2 integers per block

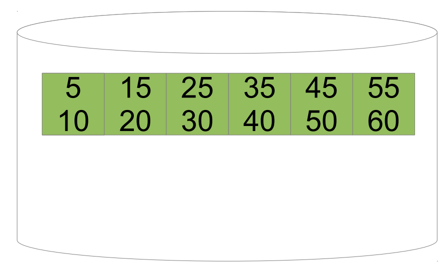
Physical Operators

Hashing R into $M-1$ buckets using M buffers



M = 3

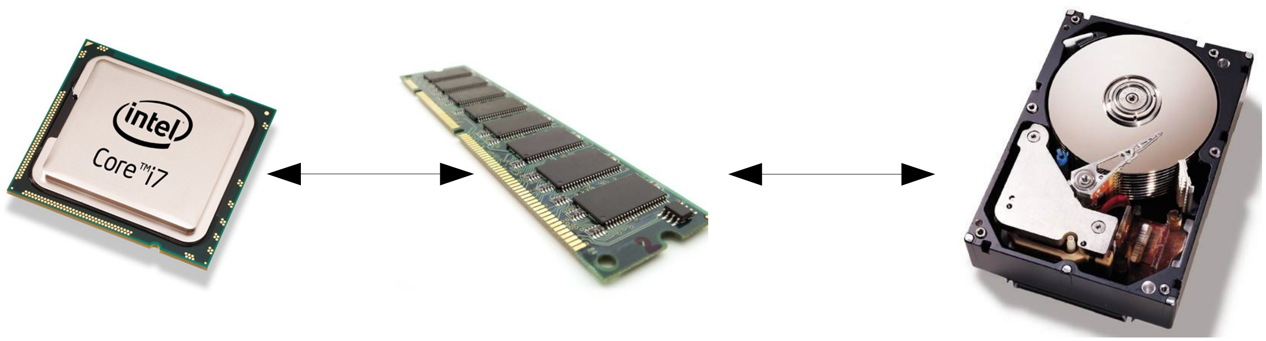




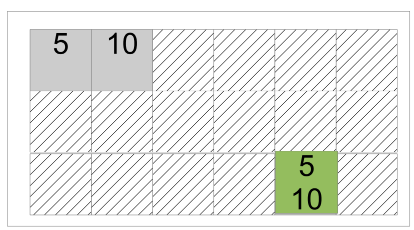
Relation R = green
2 integers per block

Physical Operators

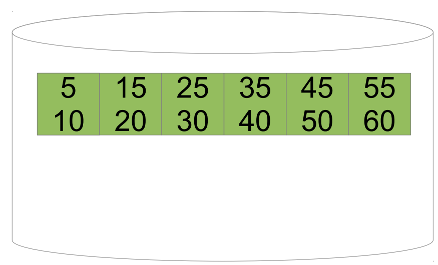
Hashing R into $M-1$ buckets using M buffers



M = 3



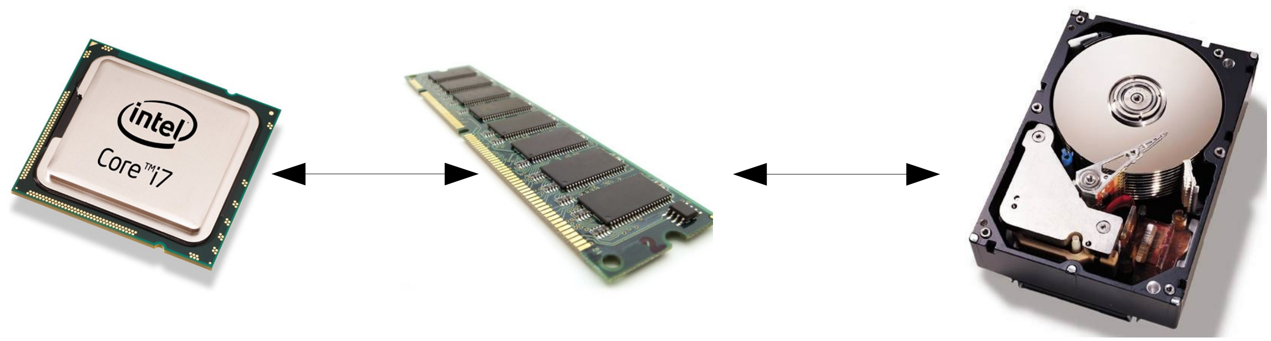
= occupied frame
= free frame



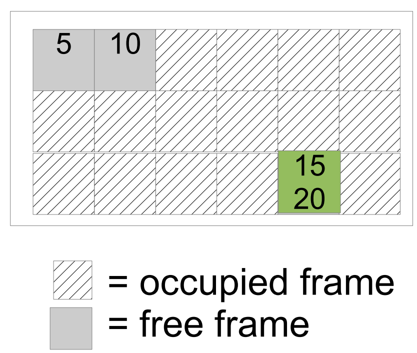
Relation R = green
2 integers per block

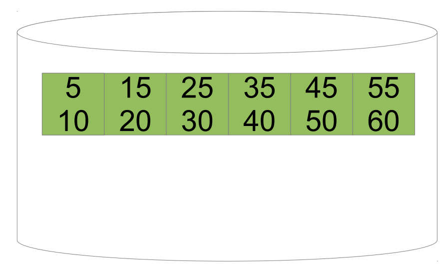
Physical Operators

Hashing R into $M-1$ buckets using M buffers



M = 3

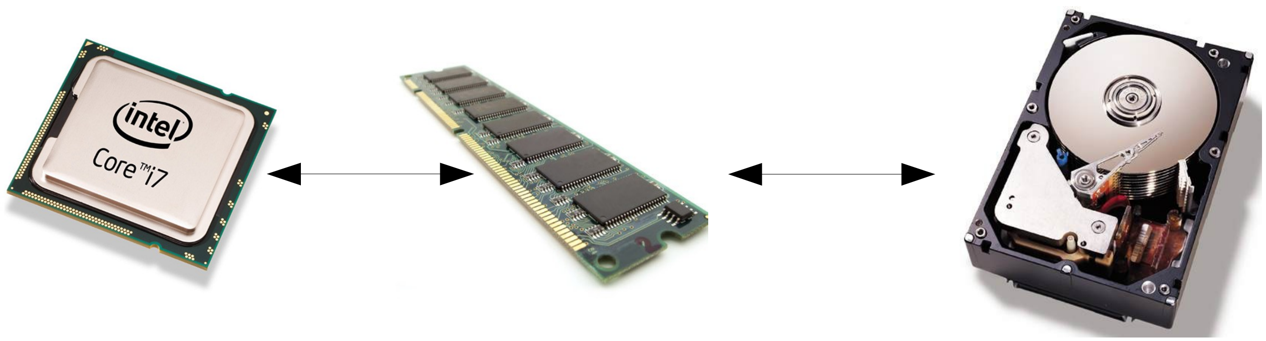




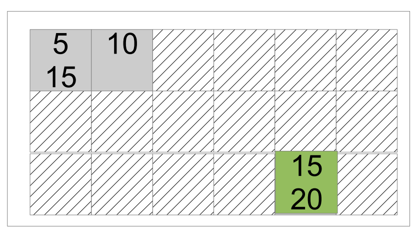
Relation R = green
2 integers per block

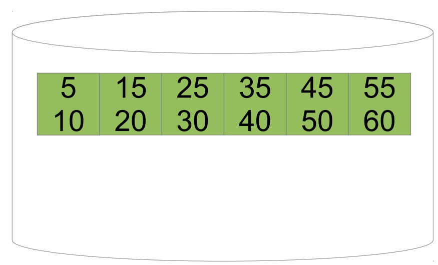
Physical Operators

Hashing R into $M-1$ buckets using M buffers



M = 3



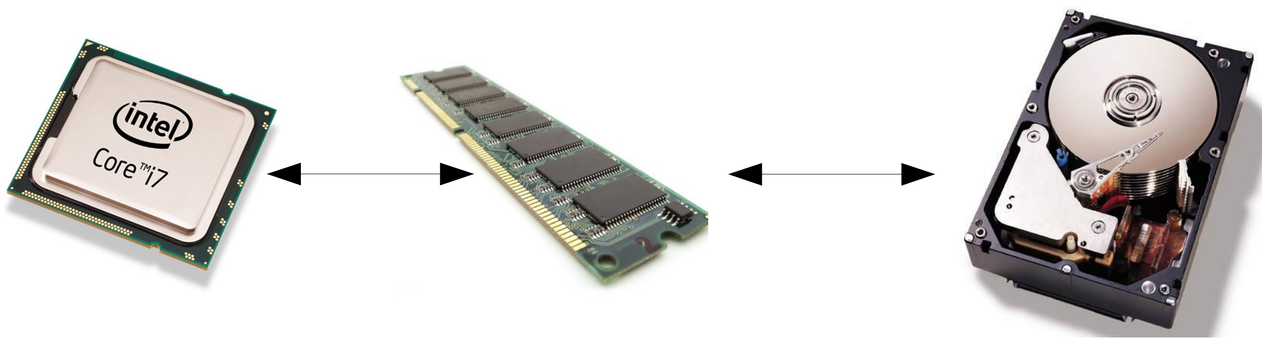


= occupied frame
= free frame

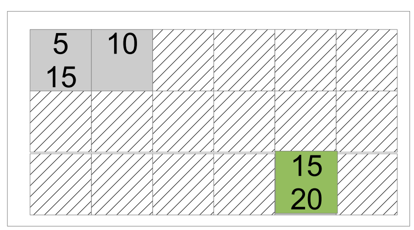
Relation R = green
2 integers per block

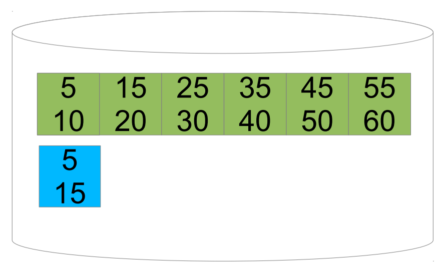
Physical Operators

Hashing R into $M-1$ buckets using M buffers



M = 3



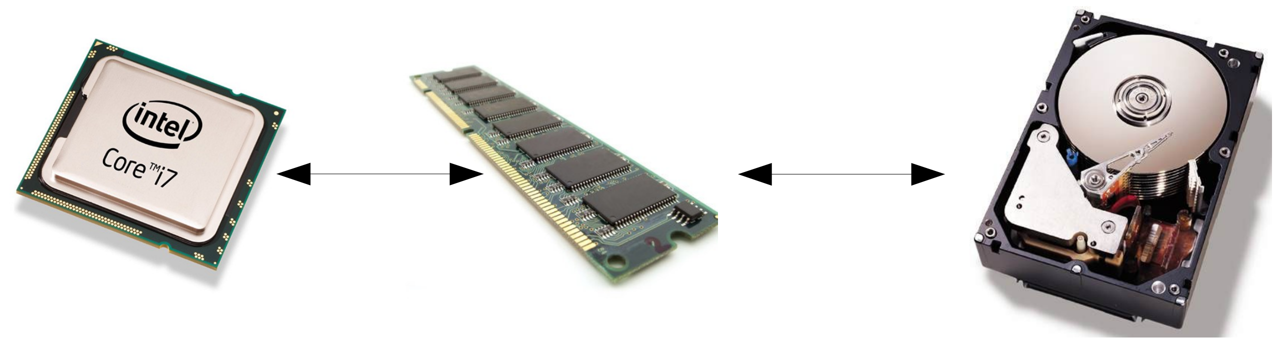


= occupied frame
= free frame

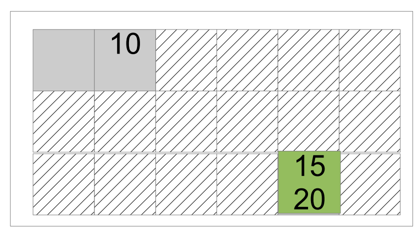
Relation R = green
2 integers per block
Bucket 1 = Blue

Physical Operators

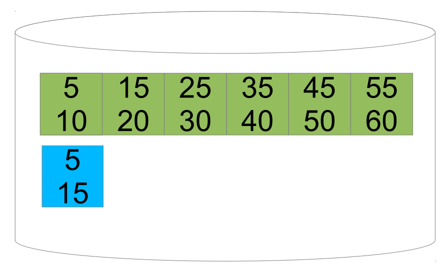
Hashing R into $M-1$ buckets using M buffers



M = 3



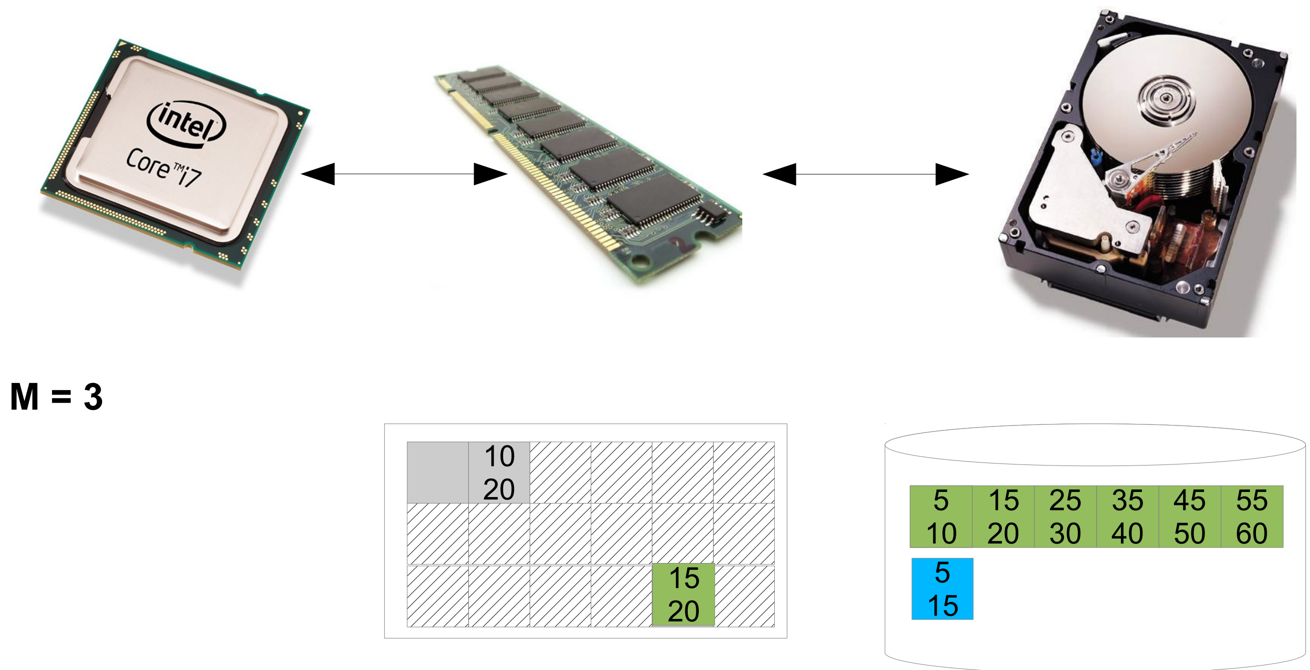
= occupied frame
= free frame



Relation R = green
2 integers per block
Bucket 1 = Blue

Physical Operators

Hashing R into $M - 1$ buckets using M buffers

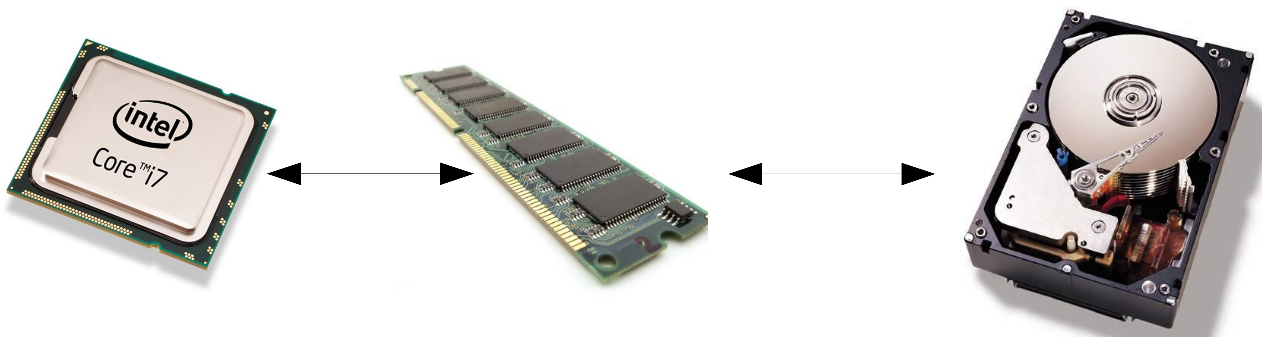


= occupied frame
= free frame

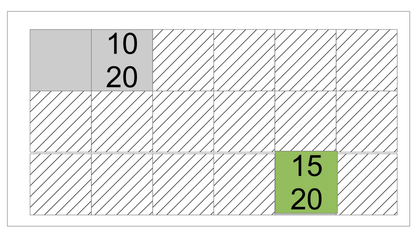
Relation R = green
2 integers per block
Bucket 1 = Blue

Physical Operators

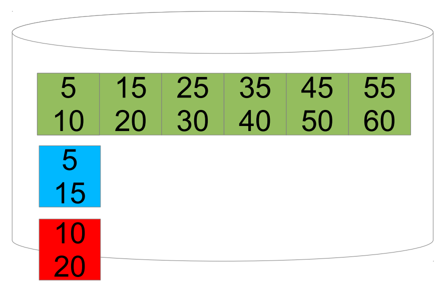
Hashing R into $M-1$ buckets using M buffers



M = 3



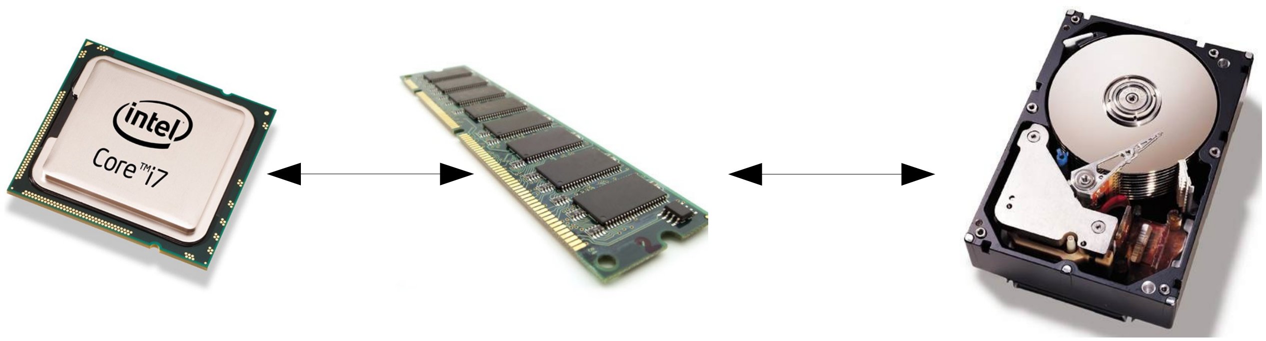
= occupied frame
= free frame



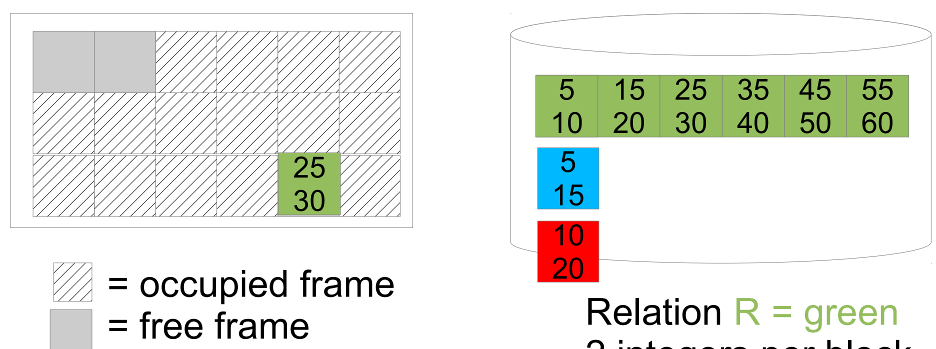
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

Physical Operators

Hashing R into $M-1$ buckets using M buffers



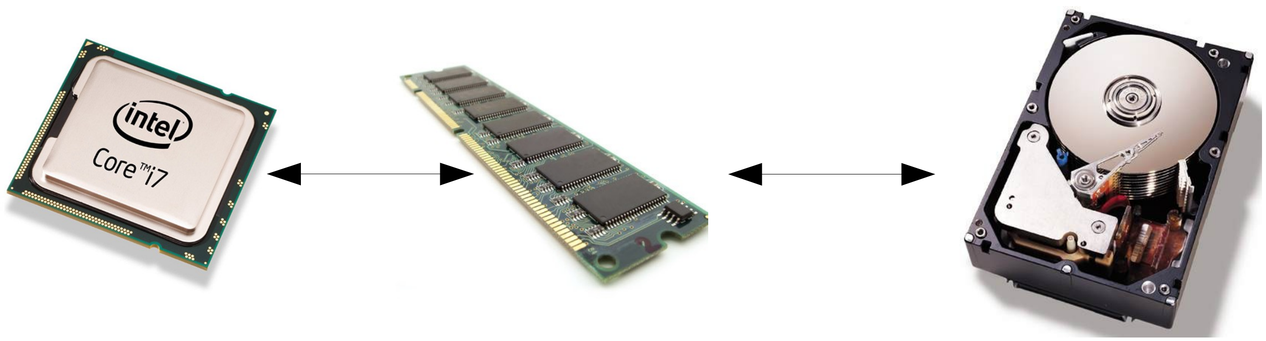
M = 3



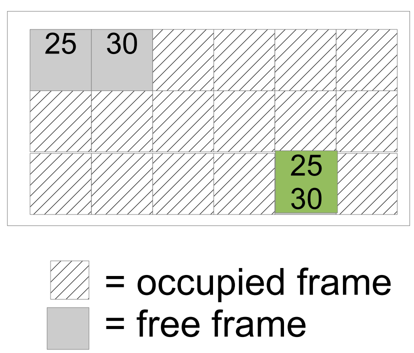
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

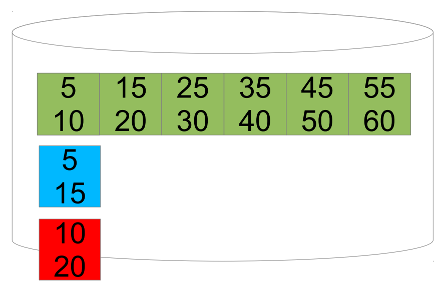
Physical Operators

Hashing R into $M-1$ buckets using M buffers



M = 3

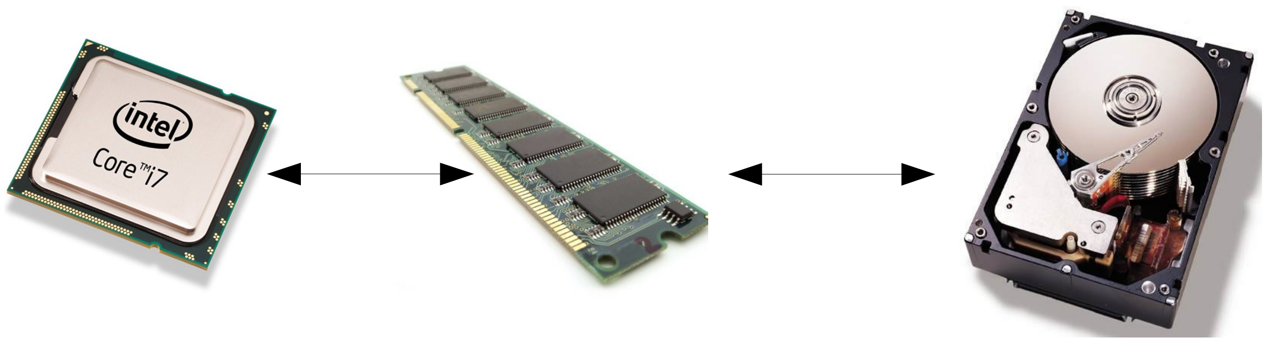




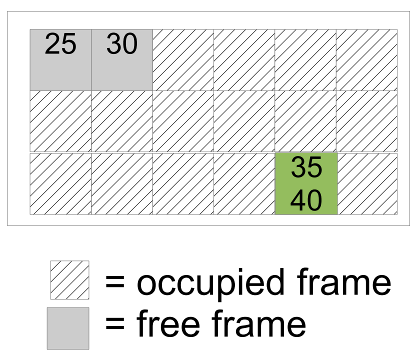
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

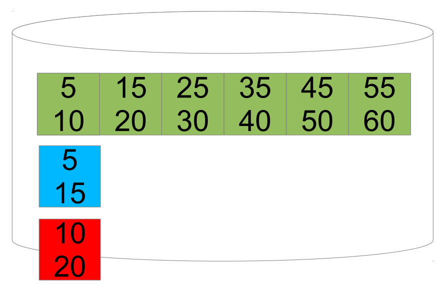
Physical Operators

Hashing R into $M - 1$ buckets using M buffers



M = 3

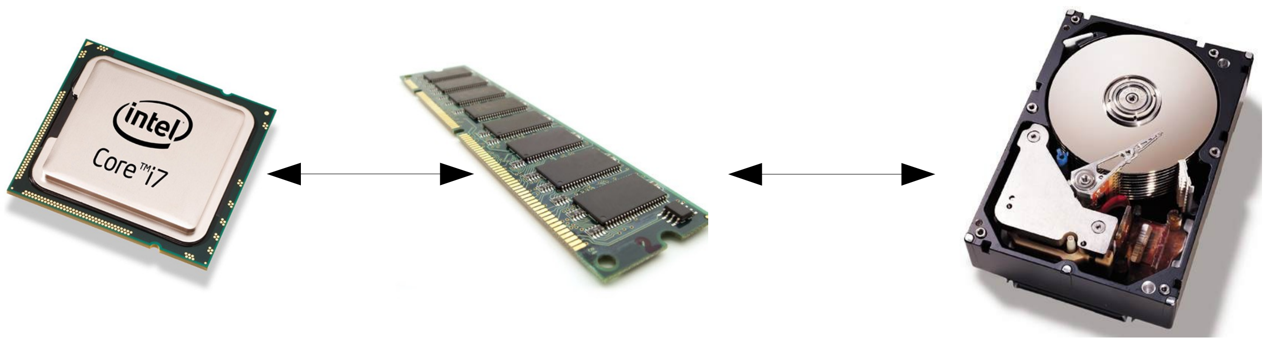




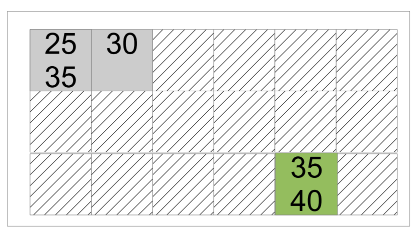
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

Physical Operators

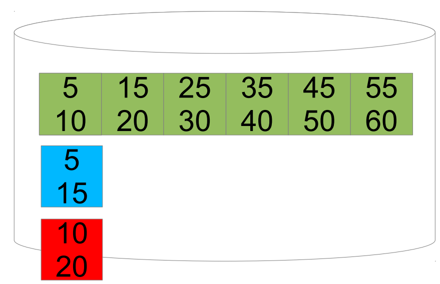
Hashing R into $M-1$ buckets using M buffers



M = 3



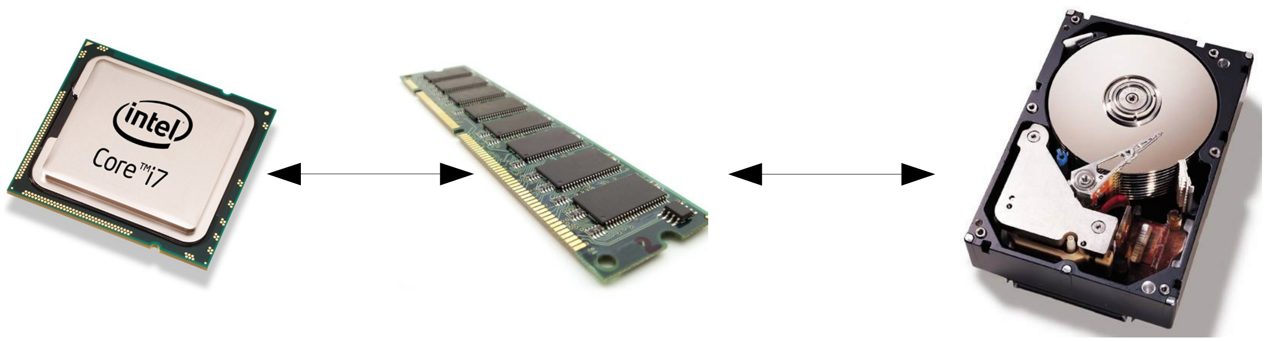
= occupied frame
= free frame



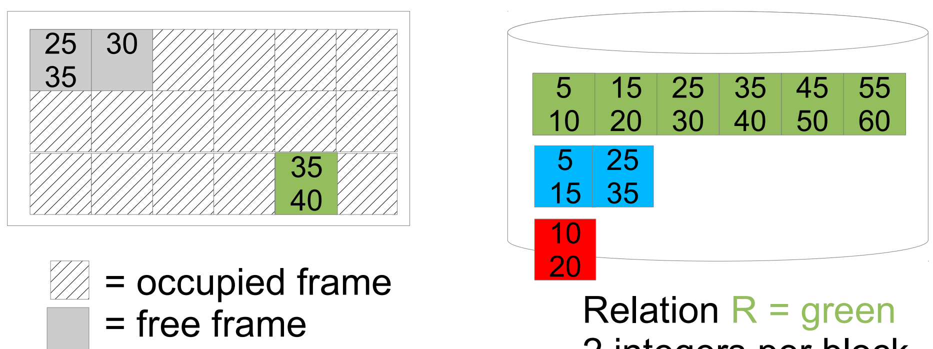
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

Physical Operators

Hashing R into $M - 1$ buckets using M buffers



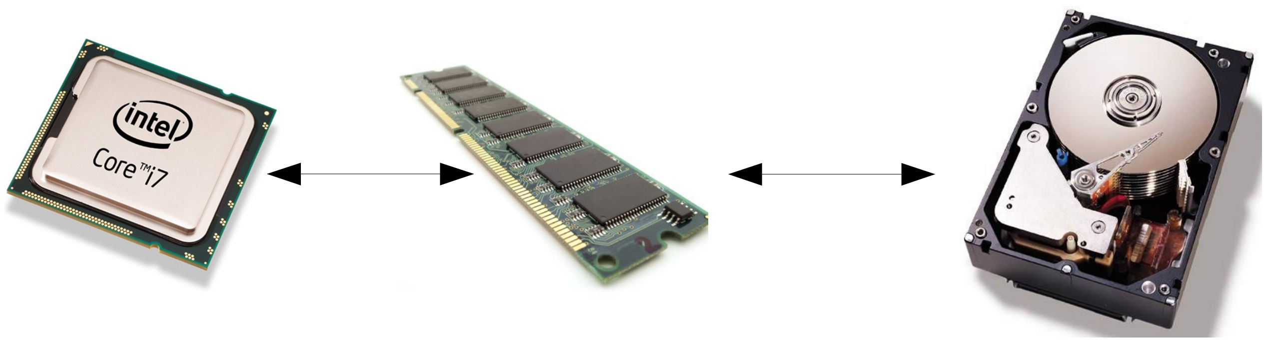
M = 3



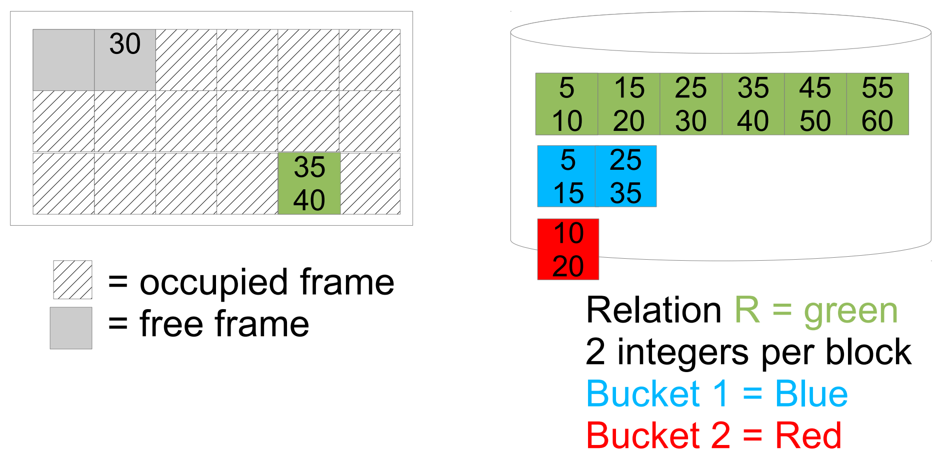
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

Physical Operators

Hashing R into $M-1$ buckets using M buffers

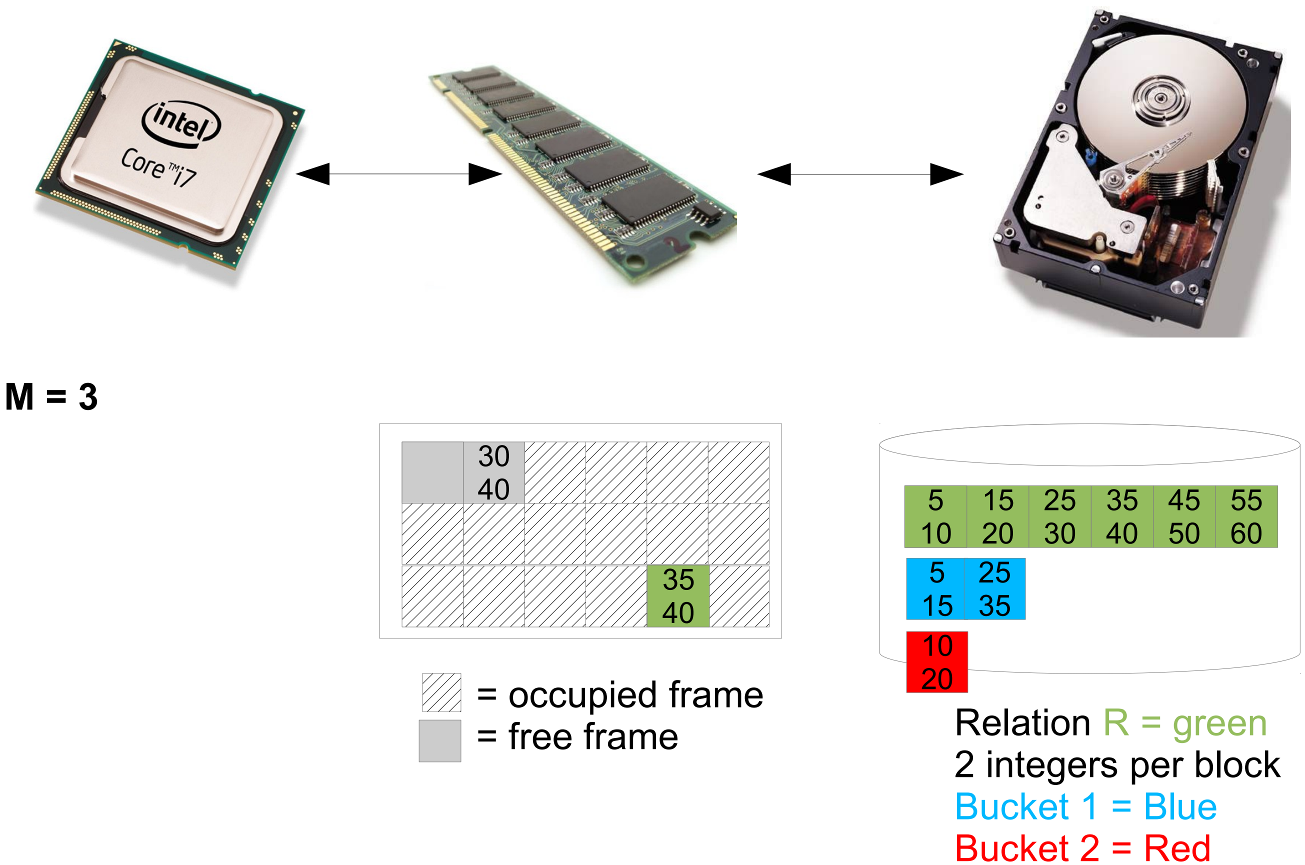


M = 3



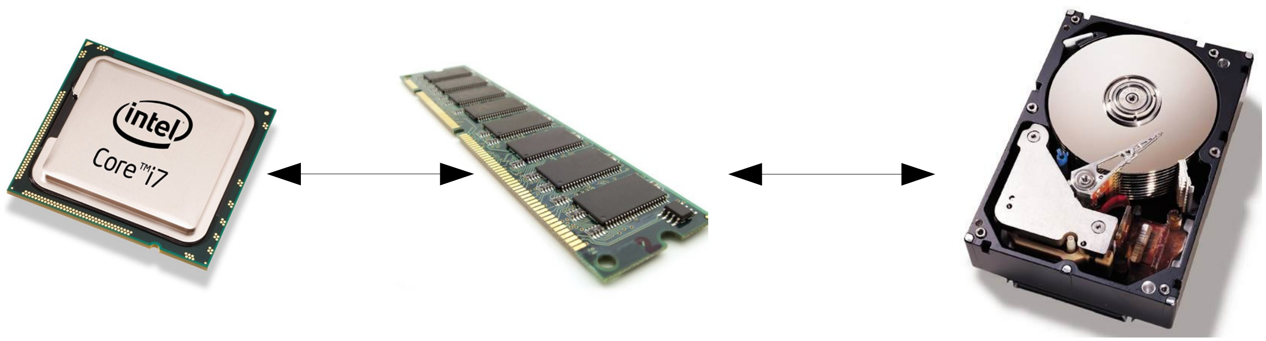
Physical Operators

Hashing R into $M-1$ buckets using M buffers

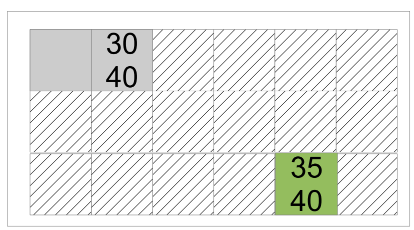


Physical Operators

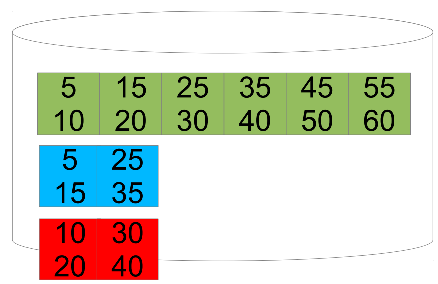
Hashing R into $M-1$ buckets using M buffers



M = 3



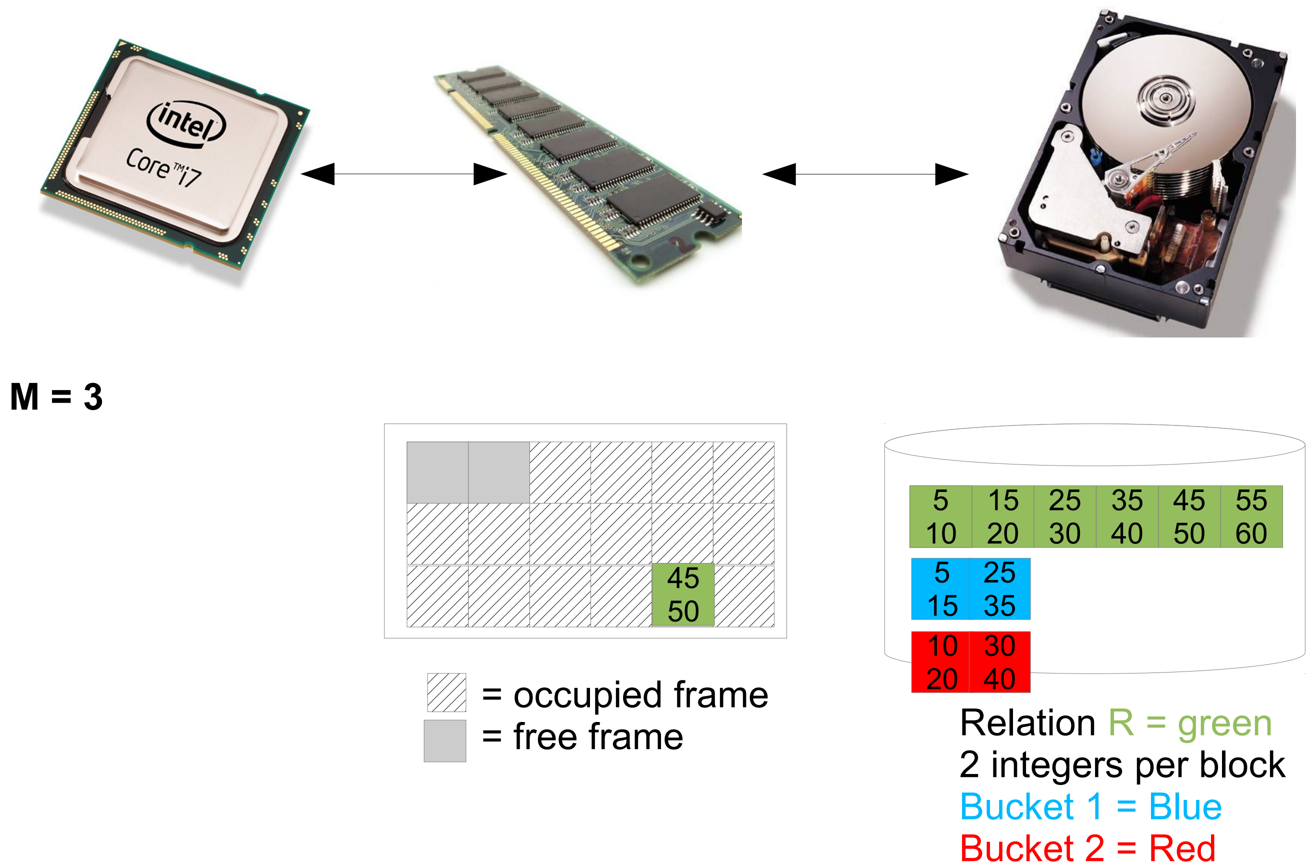
= occupied frame
= free frame



Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

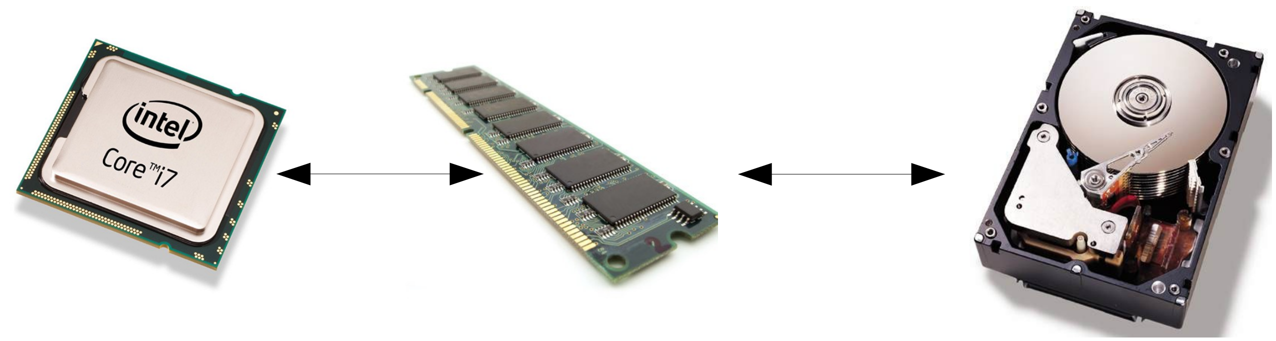
Physical Operators

Hashing R into $M-1$ buckets using M buffers

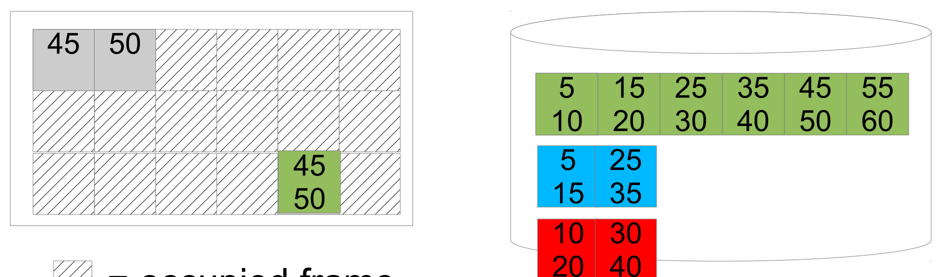


Physical Operators

Hashing R into $M-1$ buckets using M buffers



M = 3

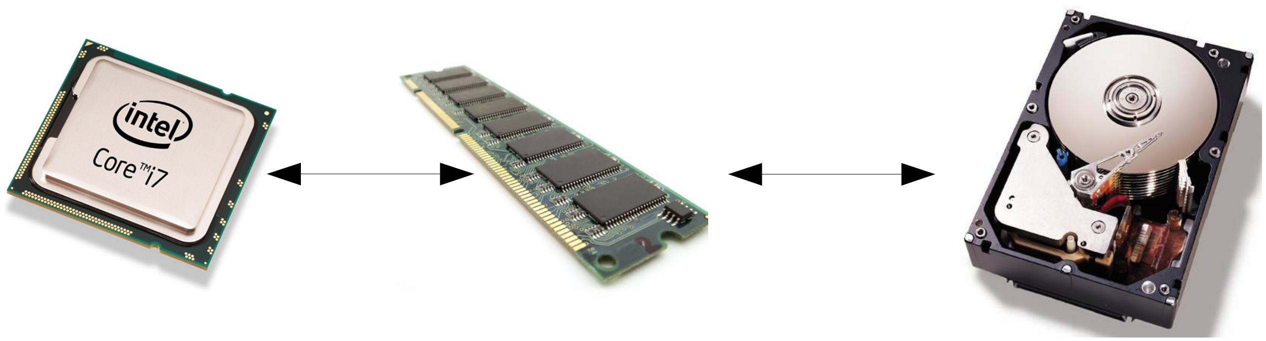


= occupied frame
= free frame

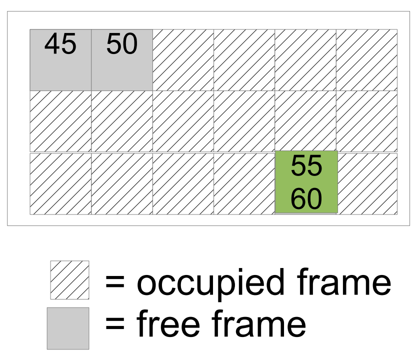
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

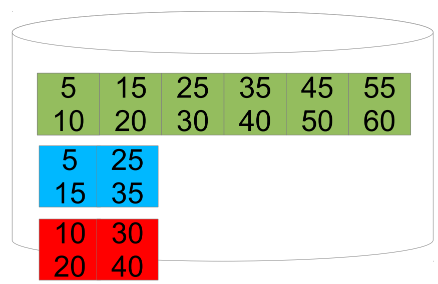
Physical Operators

Hashing R into $M-1$ buckets using M buffers



M = 3

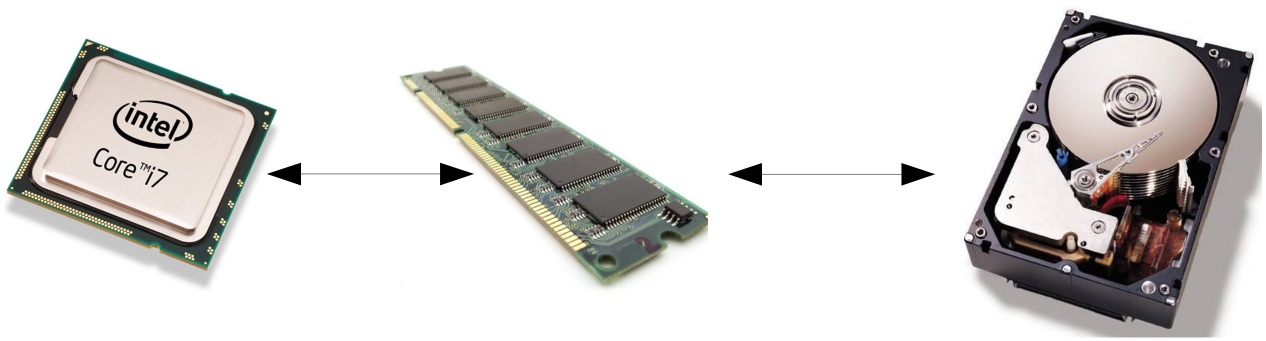




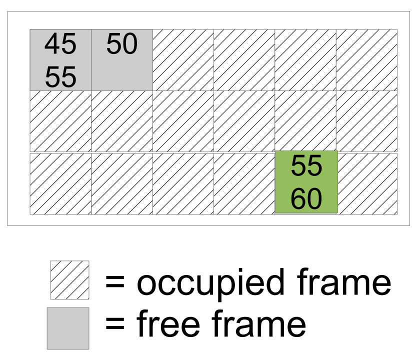
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

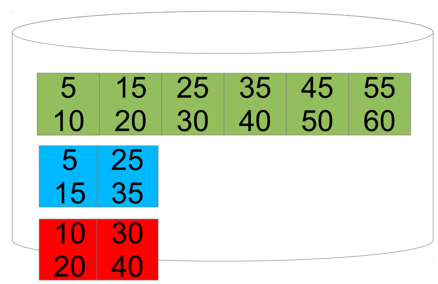
Physical Operators

Hashing R into $M-1$ buckets using M buffers



M = 3

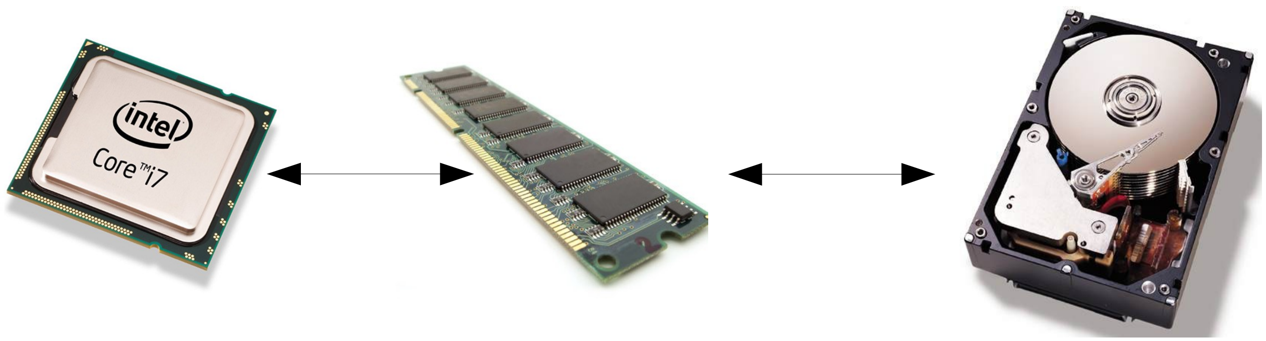




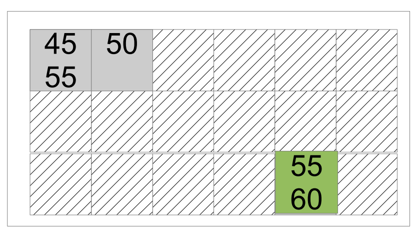
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

Physical Operators

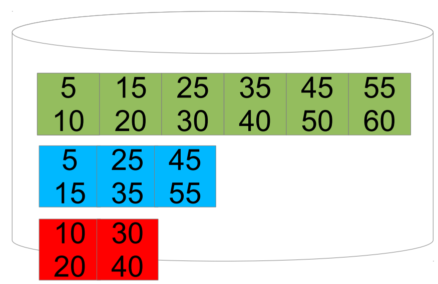
Hashing R into $M-1$ buckets using M buffers



M = 3



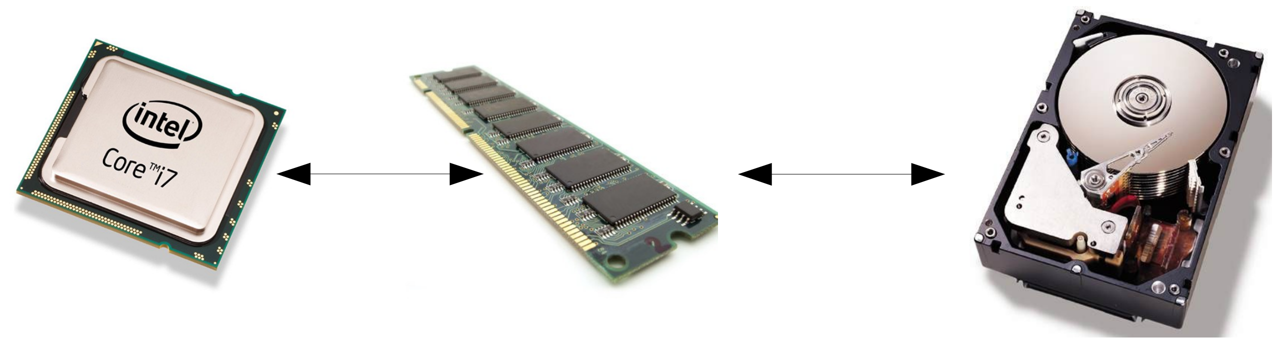
= occupied frame
= free frame



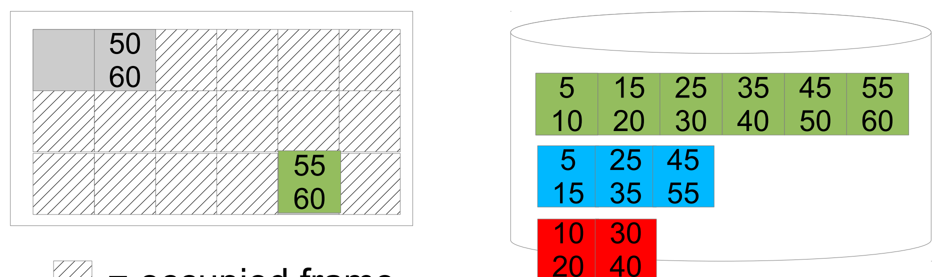
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

Physical Operators

Hashing R into $M-1$ buckets using M buffers



M = 3

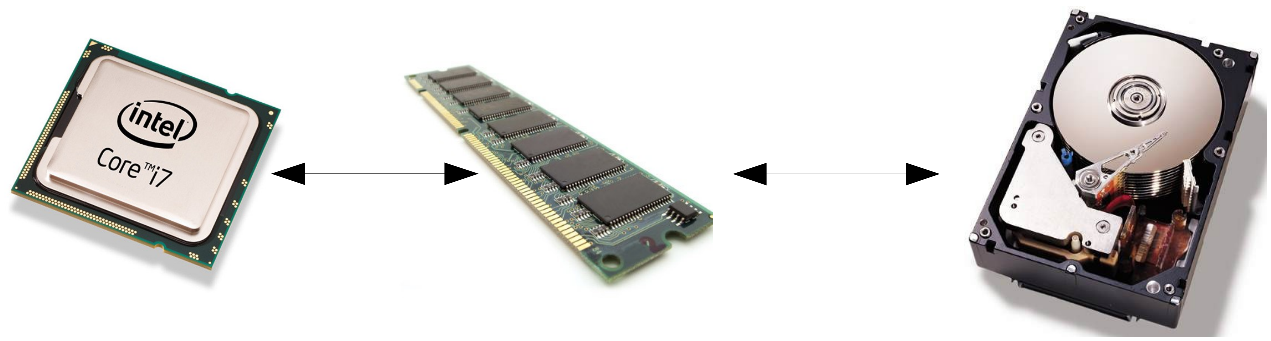


= occupied frame
= free frame

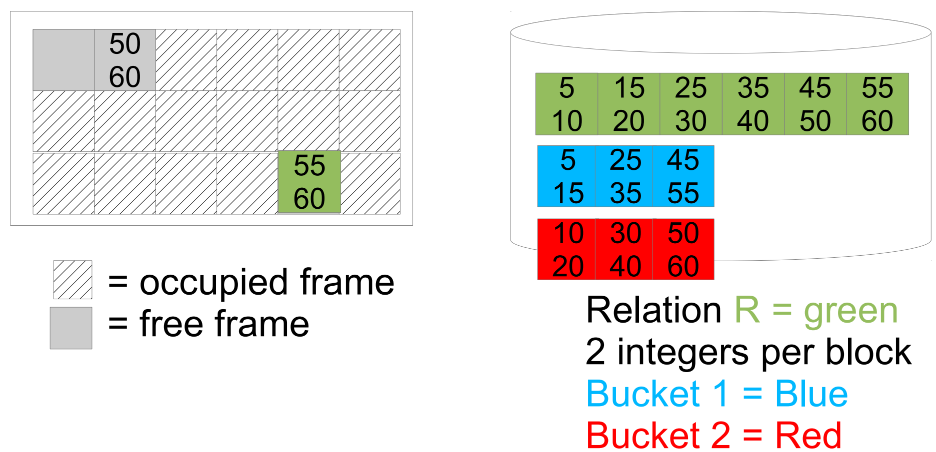
Relation R = green
2 integers per block
Bucket 1 = Blue
Bucket 2 = Red

Physical Operators

Hashing R into $M-1$ buckets using M buffers



M = 3



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

Hash-based set union

- The book states that in practice one level of partitioning suffices.
- The book hence focuses on the scenario where we only need two passes: "two-pass, hash-based set union" and only sketches the generalization to multiple passes.

  The algorithm is called two-pass because we need 1 pass through the data to partition it, and another one to do the pairwise single-pass union of the buckets
- Under the assumption that one level of partitioning suffices, our cost formula hence specializes to the cost: $3B(R) + 3B(S)$
- But: one level of partitioning only suffices if $\frac{B(R)}{M-1} \leq M - 1$, or (approximately) $B(R) \leq M^2$ (where R is the smaller relation of R and S)

  $\rightarrow$ These are the formulas introduced in the book!

Physical Operators

Other operations on relations

To compute (bag) intersection and (bag) difference we can modify the previous algorithms. The costs remain the same

Also the removal of duplicates can be done using the same techniques.

$\rightarrow$ See book!

Physical Operators

One-pass Join

Assume that $M - 1 \geq B(R)$. We can then compute $R(X,Y) \bowtie S(Y,Z)$ as follows:

load R into memory buffers $N_1, \ldots, N_{B(R)}$;
  for each block B_S in S **do**
    load B_S into buffer N_0;
    for each tuple t_S in N_0 **do**
      for each tuple matching tuple t_R in $N_1, \ldots, N_{B(R)}$ **do**
      output $t_R \bowtie t_S$

- Cost: $B(R) + B(S)$ I/O operations
- There is also the cost of finding the matching tuples of t_S 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

Nested Loop Join

We can also alternatively compute $R(X, Y) \bowtie S(Y, Z)$ as follows:

for each segment G of $M - 1$ blocks of R **do**
  load G into buffers $N_1, \ldots, N_{M-1}$;
  for each block B_S in S **do**
    load B_S into buffer N_0;
    for each tuple t_R in $N_1, \ldots, N_{M-1}$ **do**
      for each tuple t_S in N_0 **do**
        if $t_R.Y = t_S.Y$ then **output** $t_R \bowtie t_S$

Cost:

$$\boxed{B(R) + B(S) \times \frac{B(R)}{M - 1}}$$

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. $\rightarrow$ See book
- In this case the cost analysis is similar to the analysis for sort-based set union. Similarly, it is possible to optimize and gain $2B(R) + 2B(S)$ I/O operations (provided there is enough memory).
- The book also focuses on "two-pass sort-merge join".
- Remark: When R has a BTree index on Y, then it is not necessary to sort R (why?). The same holds for S.

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

- The cost analysis is the same as the analysis for hash-based set union
- Again the book focuses on "two-pass hash-join":

  one pass for the partitioning, one pass for the join

Physical Operators

Index-Join

Assume that S has an index on attribute Y. We can then alternatively compute the join $R(X,Y) \bowtie S(Y,Z)$ by searching, for every tuple t in R, the matching tuples in S (using the index).

Cost when the index on Y is not clustered:

$$B(R) + T(R) \times \lceil T(S)/V(S,Y) \rceil$$

Cost when the index on Y is clustered:

$$B(R) + T(R) \times \lceil B(S)/V(S,Y) \rceil$$

$\rightarrow$ See book

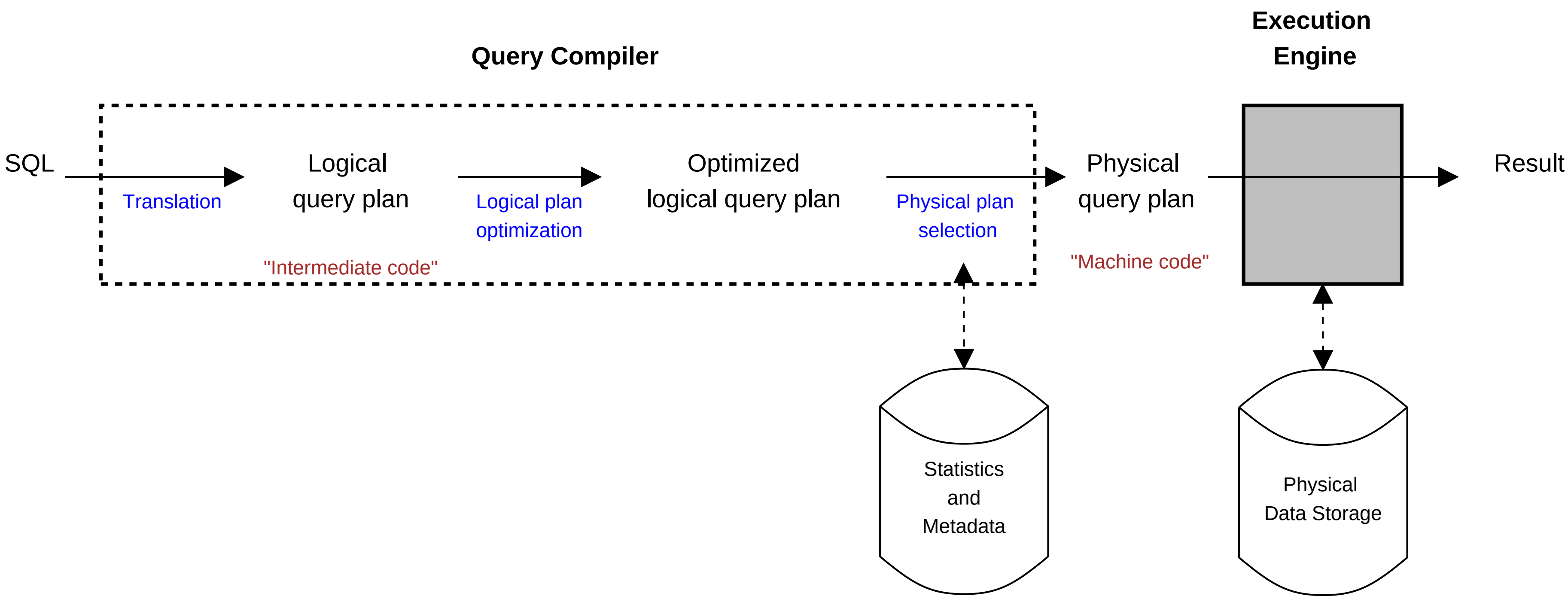
General comment

The book often omits the ceiling operations ($\lceil \cdot \rceil$) when calculating costs. In the exercises you must always include these operations!

Cost-Based Plan Selection

Enumerate, Estimate, Select

Cost-Based Plan Selection

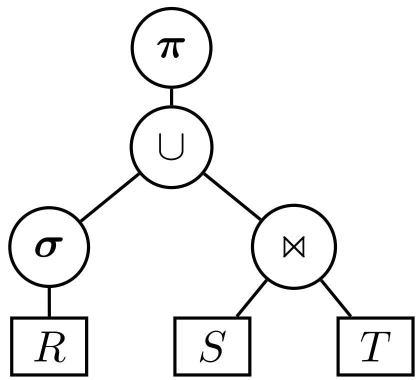


Components of the query compiler that we already know:

- SQL $\rightarrow$ relational algebra (i.e., a logical query plan)
- Logical query plan $\rightarrow$ optimized logical query plan

Cost-Based Plan Selection

The next step: logical query plan $\rightarrow$ physical query plan

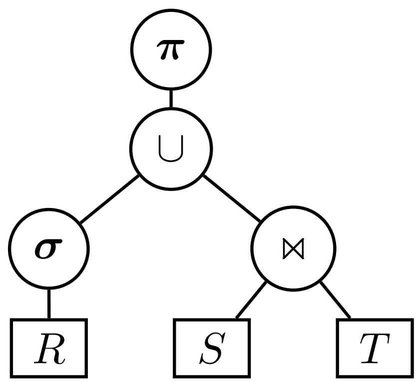


- To obtain a physical query plan we need to assign to each node in the logical query plan a physical operator.
- We want to obtain the physical plan with the smallest total execution cost.
- Hence, we need to compare, for every node and every applicable physical operator, its cost.

- In order to estimate this cost we need (among others) the parameters $B(R)$, $T(R)$, and $V(R, A_1, \ldots, A_n)$
- These belong to the statistics that a DBMS typically stores in its system catalog
- But these statistics only exist for the relations stored in the database, not for subresults computed during query evaluation!

Cost-Based Plan Selection

Result size estimation



- For every internal node n we hence need to estimate the parameters $B(n)$, $T(n)$, and $V(n, A_1, \ldots, A_k)$
- Note that we can compute $B(n)$ given (1) $T(n)$; (2) the size of the tuples output by n; and (3) the size of a block
- Also note that $T(n)$ and $V(n, A_1, \ldots, A_k)$ only depend on the logical query plan, not on the physical plan that we are computing!

Cost-Based Plan Selection

Result size estimation: projection

- General formula: $T(\pi_L(R)) = T(R)$
- Remember that our version of the projection operator is bag-based and does not remove duplicates; to remove duplicates we use the operator δ.
- While projection does not change the number of tuples, it does change the number of blocks needed to store the resulting relation, as illustrated by the following example.

Example

- $R(A, B, C)$ is a relation with A and B integers of 4 bytes each; C a string of 100 bytes. Tuple headers are 12 bytes. Blocks are 1024 bytes and have headers of 24 bytes. $T(R) = 10000$ and $B(R) = 1250$.
- Question: how many blocks do we need to store $\pi_{A,B}(R)$?

Cost-Based Plan Selection

Result size estimation: projection

- General formula: $T(\pi_L(R)) = T(R)$
- Remember that our version of the projection operator is bag-based and does not remove duplicates; to remove duplicates we use the operator δ.
- While projection does not change the number of tuples, it does change the number of blocks needed to store the resulting relation, as illustrated by the following example.

Example

- $R(A, B, C)$ is a relation with A and B integers of 4 bytes each; C a string of 100 bytes. Tuple headers are 12 bytes. Blocks are 1024 bytes and have headers of 24 bytes. $T(R) = 10000$ and $B(R) = 1250$.
- **Answer:** resulting records need to record the header + A-field + B-field. The size of these records is hence $12 + 4 + 4 \quad = 20$ bytes. We can hence store $(1024 - 24)/20 = 50$ tuples in one block. Thus $B(\pi_{A,B}(R)) = T(\pi_{A,B}(R))/50 = 10000/50 = 200$ blocks.

Cost-Based Plan Selection

Result size estimation: selection $\sigma_P(R)$ with P a filter predicate

- General formula:
  $$\boxed{T(\sigma_P(R)) = T(R) \times \textit{sel}_P(R)}$$
  where $\textit{sel}_P(R)$ is the estimated fraction of tuples in R that satisfy predicate P.

- In other words, $\textit{sel}_P(R)$ is the estimated probability that a tuple in R satisfies P.
- $\textit{sel}_P(R)$ is usually called the selectivity of filter predicate P.
- How we calculate $\textit{sel}_P(R)$ depends on what P is.

Cost-Based Plan Selection

Result size estimation: selection $\sigma_{A=c}(R)$ with c a constant

$$\boxed{\mathit{sel}_{A=c}(R) = \frac{1}{V(R,A)}}$$

- Intuition: there are $V(R,A)$ distinct A-values in R. Assuming that A-values are uniformly distributed, the probability that a tuple has A-value c is $1/V(R,A)$.
- While this intuition assumes that values are uniformly distributed, it can be shown that this selectivity is a good estimate on average, provided that c is chosen randomly.

Example

- $R(A,B,C)$ is a relation. $T(R) = 10000$. $V(R,A) = 50$.
- Then $T(\sigma_{A=10}(R))$ is estimated by:

$$T(\sigma_{A=10}(R)) = T(R) \times \frac{1}{V(R,A)} = \frac{10000}{50} = 200.$$

Cost-Based Plan Selection

Result size estimation: selection $\sigma_{A=c}(R)$ with c a constant

- Better selectivity estimates are possible if we have more detailed statistics
- A DBMS typically collects histograms that detail the distribution of values.
- Such histograms are only available for base relations, however, not for subresults!

Example

- $R(A, B, C)$ is a relation. The DBMS has collected the following equal-width histogram on A:

range	$[1, 10]$	$[11, 20]$	$[21, 30]$	$[31, 40]$	$[41, 50]$
tuples in range	50	2000	2000	3000	2950

- Then $\mathit{sel}_{A=10}(R)$ can be estimated by:

$$\mathit{sel}_{A=10}(R) = \frac{50}{10000} \times \frac{1}{10}$$

Cost-Based Plan Selection

Result size estimation: selection $\sigma_{A<c}(R)$

$$\boxed{\textit{sel}_{A<c}(R) = \tfrac{1}{2}} \quad \text{or} \quad \boxed{\textit{sel}_{A<c}(R) = \tfrac{1}{3}}$$

- This is just a heuristic, without any correctness guarantees.
- (The intuitive rationale is that queries involving an inequality tend to retrieve a small fraction of the possible tuples.)

Example

- $R(A, B, C)$ is a relation. $T(R) = 10000$.
- Then $T(\sigma_{B<10}(R))$ is estimated by:

$$T(\sigma_{B<10}(R)) = T(R) \times \frac{1}{3} = 3334.$$

Cost-Based Plan Selection

Result size estimation: selection $\sigma_{A<c}(R)$

- Again, better estimates are possible if we have more detailed statistics

Example

- $R(A, B, C)$ is a relation. $T(R) = 10000$. The DBMS statistics show that the values of the B attribute lie within the range $[8, 57]$, uniformly distributed.

- Question: what would be a reasonable estimate of $\mathit{sel}_{B<10}(R)$?

Cost-Based Plan Selection

Result size estimation: selection $\sigma_{A<c}(R)$

- Again, better estimates are possible if we have more detailed statistics

Example

- $R(A, B, C)$ is a relation. $T(R) = 10000$. The DBMS statistics show that the values of the B attribute lie within the range $[8, 57]$, uniformly distributed.
- Question: what would be a reasonable estimate of $\mathit{sel}_{B<10}(R)$?
- Answer: We see that $57 - 8 + 1$ different values of B are possible; however only records with values $B = 8$ or $B = 9$ satisfy the filter $B < 10$. Therefore,

$$\mathit{sel}_{B<10}(R) = \frac{2}{(57 - 8 + 1)} = \frac{2}{50} = 4\%$$

and hence

$$T(\sigma_{B<10}(R)) = T(R) \times \mathit{sel}_{B<10}(R) = 400.$$

Cost-Based Plan Selection

Result size estimation: selection $\sigma_{A \neq c}(R)$

$$\boxed{\textit{sel}_{A \neq c}(R) = \frac{V(R,A)-1}{V(R,A)}}$$

- Question: Can you give intuitive meaning to this formula?

Cost-Based Plan Selection

Result size estimation: selection $\sigma_{A \neq c}(R)$

$$\boxed{\textit{sel}_{A \neq c}(R) = \frac{V(R,A)-1}{V(R,A)}}$$

- Question: Can you give intuitive meaning to this formula?
- Answer: $1/V(R,A)$ is the (estimated) probability that a tuple satisfies $A = c$. Therefore
$$1 - \textit{sel}_{A=c}(R) = 1 - \frac{1}{V(R,A)} = \frac{V(R,A)-1}{V(R,A)}$$
is the (estimated) probability that a tuple does not satisfy $A = c$.

Cost-Based Plan Selection

Result size estimation: selection $\sigma_{\textbf{NOT}\ P_1}(R)$

$$\boxed{\textit{sel}_{\mathrm{NOT}\ P_1}(R) = 1 - \textit{sel}_{P_1}(R)}$$

Cost-Based Plan Selection

Result size estimation: selection $\sigma_{P_1 \text{ AND } P_2}(R)$

$$\boxed{\textit{sel}_{P_1 \text{ AND } P_2}(R) = \textit{sel}_{P_1}(R) \times \textit{sel}_{P_2}(R)}$$

- This implicitly assumes that filter predicates P_1 and P_2 are independent.
- Hence, in essence we treat $\sigma_{P_1 \text{ AND } P_2}(R)$ as $\sigma_{P_1}(\sigma_{P_2}(R))$
- The order does not matter, treating this as $\sigma_{P_2}(\sigma_{P_1}(R))$ gives the same results.

Example

- $R(A, B, C)$ is a relation. $T(R) = 10000$. $V(R, A) = 50$.
- Then we estimate $T(\sigma_{A=10 \text{ AND } B<10}(R)$ to be:

$$T(R) \times \textit{sel}_{A=10}(R) \times \textit{sel}_{B<10}(R) = T(R) \times \frac{1}{V(R, A)} \times \frac{1}{3} = 67.$$

Cost-Based Plan Selection

Result size estimation: selection $\sigma_{P_1 \text{ OR } P_2}(R)$

$$\boxed{\textit{sel}_{P_1 \text{ OR } P_2}(R) = \min(\textit{sel}_{P_1}(R) + \textit{sel}_{P_2}(R), 1)}$$

- The term $\textit{sel}_{P_1}(R) + \textit{sel}_{P_2}(R)$ implicitly assumes that filter predicates P_1 and P_2 are independent, and select disjoint sets of tuples.
- Disjointness is often not satisfied and then we count some tuples twice.
- But of course, the selectivity can never be greater than 1.
- Hence, we take the minimum of these two terms.

Cost-Based Plan Selection

Result size estimation: selection $\sigma_{P_1 \text{ OR } P_2}(R)$

More complicated: treat this as $\sigma_{\text{NOT}(\text{NOT } P_1 \text{ AND NOT } P_2)}(R))$.

$$\boxed{\mathit{sel}_{P_1 \text{ OR } P_2}(R) = 1 - (1 - \mathit{sel}_{P_1}(R)) \times (1 - \mathit{sel}_{P_2}(R))}$$

Cost-Based Plan Selection

Result size estimation: cartesian product $R \times S$

- General formula:

$$\boxed{T(R \times S) = T(R) \times T(S)}$$

Cost-Based Plan Selection

Result size estimation: natural join $R \bowtie S$

- Assume the relation schema $R(X, Y)$ and $S(Y, Z)$, i.e., we join on Y.
- Many cases are possible
  - It is possible that R and S do not have any Y value in common. In that case, $T(R \bowtie S) = 0$.
  - Y might be the key of S and a foreign key of R, so each tuple of R joins with exactly one tuple of S. Then $T(R \bowtie S) = T(R)$.
  - Almost all of the tuples of R and S could have the same Y-value. Then $T(R \bowtie S)$ is approximately $T(R) \times T(S)$.

Cost-Based Plan Selection

Result size estimation: natural join $R \bowtie S$

- Assume the relation schema $R(X, Y)$ and $S(Y, Z)$, i.e., we join on Y.
- To focus on the common cases, we make two simplifying assumptions.

1. Containment of value sets If attribute Y appears in several relations, then each relation chooses its values from a fixed list of values $y_1, y_2, y_3, \ldots$. As a consequence, if $V(R, Y) \leq V(S, Y)$ then every Y-value of R will have a joining tuple Y-value in S.

2. Preservation of value sets When joining two relations, any attribute that is not a join attribute does not lose values from its set of possible values: for such attributes $V(R \bowtie S, A) = V(R, A)$, when A is in R and $V(R \bowtie S, A) = V(S, A)$ otherwise.

Cost-Based Plan Selection

Result size estimation: natural join $R \bowtie S$

- Assume the relation schema $R(X, Y)$ and $S(Y, Z)$, i.e., we join on Y.
- Under these assumptions, we can estimate as follows.

1. **Case 1:** $V(R, Y) \leq V(S, Y)$. Then every tuple of R has $\frac{1}{V(S,Y)}$ chance of joining with a given tuple of S. Hence

$$T(R \bowtie S) = T(R) \times \frac{1}{V(S, Y)} \times T(S)$$

2. **Case 2:** $V(S, Y) \leq V(R, Y)$. Then every tuple of S has $\frac{1}{V(R,Y)}$ chance of joining with a given tuple of R. Hence

$$T(R \bowtie S) = T(R) \times \frac{1}{V(R, Y)} \times T(S)$$

Cost-Based Plan Selection

Result size estimation: natural join $R \bowtie S$

- Assume the relation schema $R(X, Y)$ and $S(Y, Z)$, i.e., we join on Y.
- Under these assumptions, we can estimate as follows.

1. **Case 1:** $V(R, Y) \leq V(S, Y)$. Then every tuple of R has $\frac{1}{V(S,Y)}$ chance of joining with a given tuple of S. Hence

$$T(R \bowtie S) = T(R) \times \frac{1}{V(S, Y)} \times T(S)$$

2. **Case 2:** $V(S, Y) \leq V(R, Y)$. Then every tuple of S has $\frac{1}{V(R,Y)}$ chance of joining with a given tuple of R. Hence

$$T(R \bowtie S) = T(R) \times \frac{1}{V(R, Y)} \times T(S)$$

General formula:

$$\boxed{T(R \bowtie S) = T(R) \times T(S) \times \frac{1}{\max(V(R,Y), V(S,Y))}}$$

Cost-Based Plan Selection

Result size estimation: natural join $R \bowtie S$

- Now assume the relation schema $R(X, Y_1, Y_2)$ and $S(Y_1, Y_2, Z)$, i.e., we join on Y_1 and Y_2.
- Under the same assumptions as before, we can estimate as follows.

Case 1: $V(R, Y_1) \leq V(S, Y_1)$ and $V(R, Y_2) \leq V(S, Y_2)$.

Then a tuple of R has $\frac{1}{V(S,Y_1)} \times \frac{1}{V(S,Y_2)}$ chance of joining with a given tuple of S. Hence

$$T(R \bowtie S) = T(R) \times \frac{1}{V(S, Y_1)} \times \frac{1}{V(S, Y_2)} \times T(S)$$

Cost-Based Plan Selection

Result size estimation: natural join $R \bowtie S$

- Now assume the relation schema $R(X, Y_1, Y_2)$ and $S(Y_1, Y_2, Z)$, i.e., we join on Y_1 and Y_2.
- Under the same assumptions as before, we can estimate as follows.

Case 2: $V(S, Y_1) \leq V(R, Y_1)$ and $V(S, Y_2) \leq V(R, Y_2)$.

Then a tuple of S has $\frac{1}{V(R,Y_1)} \times \frac{1}{V(R,Y_2)}$ chance of joining with a given tuple of R. Hence

$$T(R \bowtie S) = T(R) \times \frac{1}{V(R, Y_1)} \times \frac{1}{V(R, Y_2)} \times T(S)$$

Cost-Based Plan Selection

Result size estimation: natural join $R \bowtie S$

- Now assume the relation schema $R(X, Y_1, Y_2)$ and $S(Y_1, Y_2, Z)$, i.e., we join on Y_1 and Y_2.
- Under the same assumptions as before, we can estimate as follows.

Case 3: $V(R, Y_1) \leq V(S, Y_1)$ and $V(S, Y_2) \leq V(R, Y_2)$.

Then a tuple of R has $\frac{1}{V(S,Y_1)} \times \frac{1}{V(R,Y_2)}$ chance of joining with a given tuple of S. Hence

$$T(R \bowtie S) = T(R) \times \frac{1}{V(S, Y_1)} \times \frac{1}{V(R, Y_2)} \times T(S)$$

Cost-Based Plan Selection

Result size estimation: natural join $R \bowtie S$

- Now assume the relation schema $R(X, Y_1, Y_2)$ and $S(Y_1, Y_2, Z)$, i.e., we join on Y_1 and Y_2.
- Under the same assumptions as before, we can estimate as follows.

Case 4: $V(S, Y_1) \leq V(R, Y_1)$ and $V(R, Y_2) \leq V(S, Y_2)$.

Then a tuple of R has $\frac{1}{V(R,Y_1)} \times \frac{1}{V(S,Y_2)}$ chance of joining with a given tuple of S. Hence

$$T(R \bowtie S) = T(R) \times \frac{1}{V(R, Y_1)} \times \frac{1}{V(S, Y_2)} \times T(S)$$

Cost-Based Plan Selection

Result size estimation: natural join $R \bowtie S$

- Now assume the relation schema $R(X, Y_1, Y_2)$ and $S(Y_1, Y_2, Z)$, i.e., we join on Y_1 and Y_2.

- General formula:

$$\boxed{T(R \bowtie S) = \frac{T(R) \times T(S)}{\max(V(R,Y_1), V(S,Y_1)) \max(V(R,Y_2), V(S,Y_2))}}$$

- This generalizes straightforwardly to the case where we are joining on more than 2 attributes.

Cost-Based Plan Selection

Result size estimation

- Intersection $R \cap S$, Difference $R - S$, duplicate elimination $\delta(R)$, Grouping and aggregation $\gamma(R)$

  $\rightarrow$ see section 16.4 in the book

- A DBMS often also collects more detailed statistics

  $\rightarrow$ see sections 16.5.1 and 16.5.2 in the book

- As should be clear by now, result size estimation is not an exact art

- For commercial DBMSs, the software component that estimates result sizes is intricate and advanced!

Cost-Based Plan Selection

Join ordering

During the optimization of the logical query plan we:

- remove redundant joins;
- push selections and projections; recognize joins.

The order in which the joins are to be executed is not yet fixed, however!

Cost-Based Plan Selection

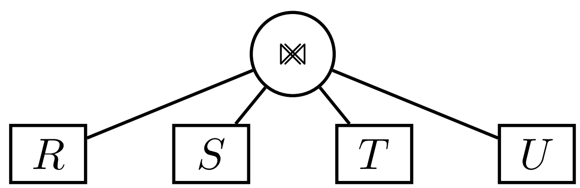
Join ordering

Example: relations $R(A, B), S(B, C), T(C, D), U(D, A)$ and the query

SQL:
```
SELECT * FROM R,S,T,U
WHERE R.B = S.B AND S.C = T.C AND T.D = U.D
```

Algebra: $R \bowtie S \bowtie T \bowtie U$

- So far, we have always considered the join as a polyadic operator:



  After all, the join order is irrelevant for logical query plans.
- However, physical join operators are binary!
- When devising a physical query plan, the join order therefore becomes very important, as we illustrate next.

Cost-Based Plan Selection

Join ordering

Example: relations $R(A, B), S(B, C), T(C, D), U(D, A)$ and the query

SQL:
```
SELECT * FROM R,S,T,U
WHERE R.B = S.B AND S.C = T.C AND T.D = U.D
```

Algebra: $R \bowtie S \bowtie T \bowtie U$

We can interpret this as:

$$((R \bowtie S) \bowtie T) \bowtie U \quad \text{or} \quad (R \bowtie S) \bowtie (T \bowtie U) \quad \text{or} \quad \ldots$$

But also as:

$$((R \bowtie T) \bowtie U) \bowtie S \quad \text{or} \quad ((R \bowtie S) \bowtie U) \bowtie T \quad \text{or} \quad \ldots$$

Cost-Based Plan Selection

Join ordering

The chosen order can influence the total cost of the physical query plan.

Consider, for example, $R(A,B), S(B,C), T(A,E)$. Assume

$$B(R) = 50 \qquad B(S) = 50 \qquad B(T) = 50$$
$$B(R \bowtie S) = 150 \qquad B(S \bowtie T) = 2500 \qquad B(R \bowtie T) = 200$$

Further assume that we execute all joins by means of the one-pass algorithm. What is the best order to compute $R \bowtie S \bowtie T$?

1. Cost of $R \bowtie (S \bowtie T)$:

$$B(R) + B(S \bowtie T) + B(S) + B(T) = 2650$$

2. Cost of $S \bowtie (R \bowtie T)$:

$$B(S) + B(R \bowtie T) + B(R) + B(T) = 350$$

3. Cost of $T \bowtie (R \bowtie S)$:

$$B(T) + B(R \bowtie S) + B(R) + B(S) = 300$$

Cost-Based Plan Selection

Join ordering

- To obtain the physical plan with the least cost we would hence have to enumerate and compare every possible join ordering.
- The number of possible orderings to join n relations is $n! \times T(n)$:
  - There are $n!$ ways to order the relations to join
  - Given a fixed ordering, there are $T(n)$ ways to create a binary tree over n leaf nodes, where

$$T(1) = 1 \qquad\qquad T(n) = \sum_{i=1}^{n-1} T(i) \times T(n-i)$$

Cost-Based Plan Selection

Join ordering

- The resulting search space is enormous.

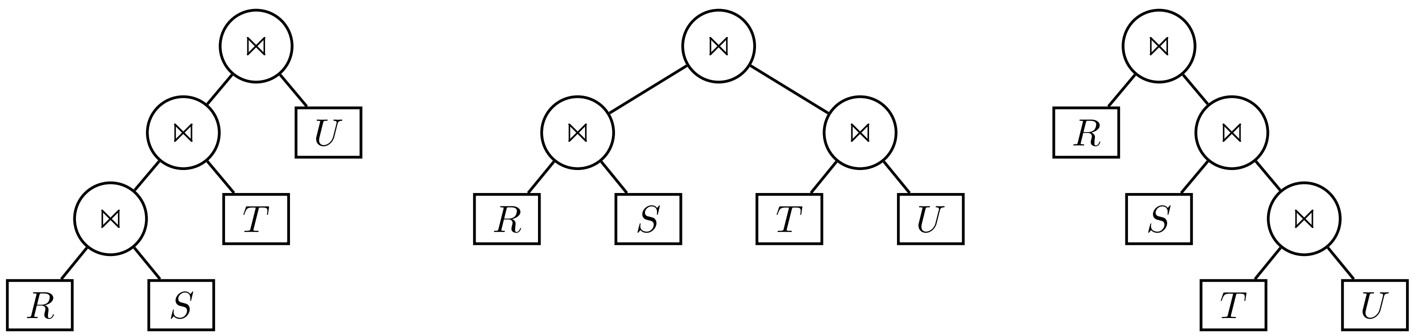
Number of relations n	$n! \times T(n)$
2	2
3	12
4	120
5	1,680
6	30,240
7	665,580
8	17,297,280

- For each of these plans, we have to consider all possible assignments of physical join algorithms to logical join operators to get the plan with the least cost.

  $\rightarrow$ Query optimization should in no case take more time than the actual execution of the query. We will therefore not consider all possible orders, but only a limited subclass.

Cost-Based Plan Selection

Kinds of join orderings



left-deep bushy right-deep

In practice a query compiler usually only considers left-deep join orderings:

- There are still $n!$ possible orderings of this form, but that is already a lot less.
- Left-deep orderings use, in general, less memory. Furthermore, in general they require fewer subresults to be stored.

$\rightarrow$ See section 16.6.3 in the book

Cost-Based Plan Selection

Plan selection

To compute the best physical plan for a given logical query plan we should, in principle:

1. Calculate all possible (left-deep) join orderings of the logical plan
2. For each such plan calculate all possible assignments of physical operators to the nodes
3. From this enormous pile of candidate physical query plans choose the one with the least estimated cost.

There are exponentially many candidate physical query plans

- Query compilation should in no case take longer than the actual execution of the query!
- In general it is hence impossible to inspect all candidate physical plans.

  Heuristics: Branch-and-Bound Plan Enumeration; Hill Climbing; Dynamic Programming; Selinger-Style; Greedy

  → See section 16.5.4, 16.6.4 and 16.6.5 in the book

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.
- 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 as our cost metric instead of the size of the intermediate results 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

Greedy plan selection need not return the optimal plan

- It may return a more expensive join ordering. For example:

$$R(A, B) \bowtie S(B, C) \bowtie T(C, D) \bowtie U(A, D)$$

Assume: the greedy algorithm computes $((R \bowtie S) \bowtie T) \bowtie U)$ with

$$B(R \bowtie S) = 100 \qquad\qquad B((R \bowtie S) \bowtie T) = 2000$$

Assume: the alternative ordering $((R \bowtie U) \bowtie T) \bowtie S$ yields

$$B(R \bowtie U) = 200 \qquad\qquad B((R \bowtie U) \bowtie T) = 1000$$

When we hence execute the joins using the one-pass algorithm we get the following costs, respectively:

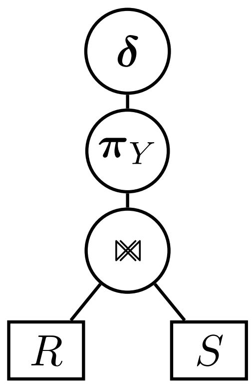
1. $B(R) + B(S) + B(R \bowtie S) + B(T) + B((R \bowtie S) \bowtie T) + B(U)$
2. $B(R) + B(U) + B(R \bowtie U) + B(T) + B((R \bowtie U) \bowtie T) + B(S)$

The second ordering yields a saving of 900 I/Os.

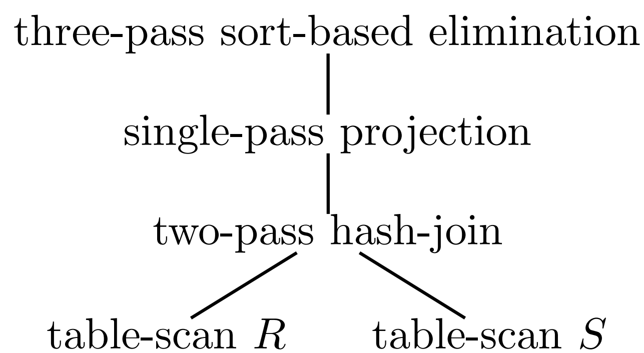
Cost-Based Plan Selection

Greedy plan selection need not return the optimal plan

- It does not take into account the properties of the output of an operator. For example (R and S share only the Y attribute):



logical plan



physical plan

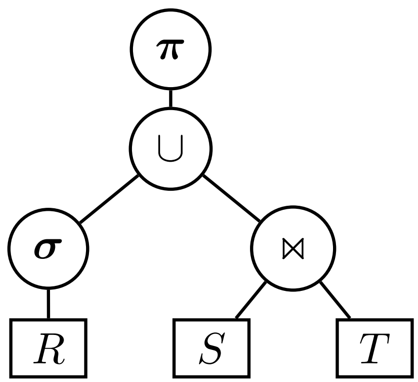
Consider the setting where there is limited memory available. The optimized sort-merge join is not applicable; only the non-optimized version. In this case the two-pass hash-join is cheaper, and is hence selected by the greedy algorithm. Because the output of $R \bowtie S$ is large, we will eventually have to remove duplicates by means of a three-pass algorithm.

If, however, we had executed the join by means of a two-pass sort-merge join, then its result would have been sorted on Y and we would have been able to compute the duplicate removal by means of the one-pass algorithm instead of the three-pass one. In that case, the total costs would have been smaller (check this!)

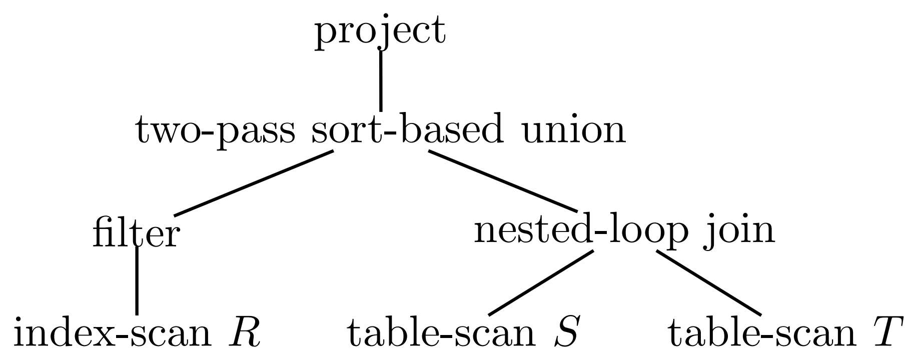
Cost-Based Plan Selection

Finally

The result of the greedy algorithm is an execution tree in which every node is a physical operator.



logical plan



physical plan

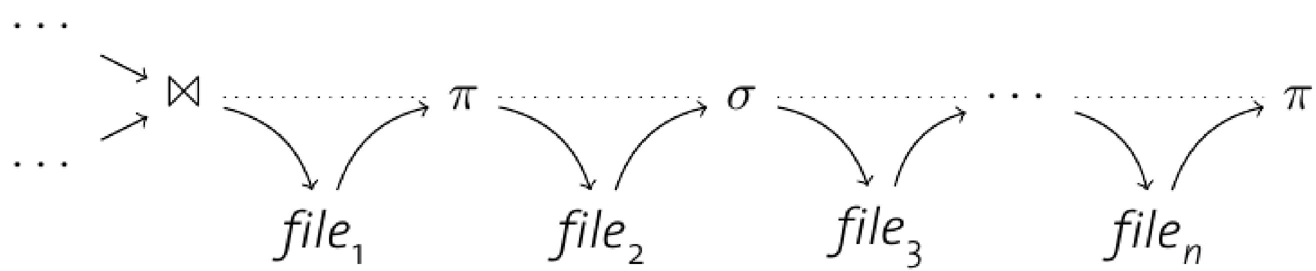
We remain to decide, for every internal node, whether we will materialize or pipeline the subresults.

$\rightarrow$ See sections 16.7.3, 16.7.4, and 16.7.5

Cost-Based Plan Selection

Pipelining versus materialization

So far, we have assumed that all database operators consume items on disk, and produce their result on disk.



- This causes a lot of I/O.
- In addition, we suffer from long response times since an operator cannot start computing its result before **all** of its inputs are fully generated ("materialized")

Cost-Based Plan Selection

Pipelining versus materialization

Alternatively, each operator could pass its result directly to the next operator. This is called pipelining.

When executed in a pipelined manner, an operator

- Starts computing results as early as possible, i.e., as soon as enough input data is available to start producing output.
- Doesn't wait until the entire output is computed, but propagates its output immediately.

The granularity in which data is passed may influence performance:

- Small chunks yield better system response time.
- Large chunks may improve the effectiveness of caches.
- Most often, data is passed a tuple at a time.

Cost-Based Plan Selection

Examples of operators that can be pipelined

- projection
- selection
- renaming
- bag-based union
- merge-joins for which the input are already known to be sorted

Cost-Based Plan Selection

Pipelining versus materialization

Pipelining reduces memory requirements and response times since each chunk of its input is propagated to the output immediately.

Some operators cannot be implemented in such a way:

- operators based on (external) sorting (i.e. sort-merge join)
- operators based on external hashing (i.e., hash join)
- grouping and duplicate elimination over unsorted input

Operators that cannot be pipelined are said to be blocking

- Blocking operators consume their entire input before they can produce any output.
- Their data is typically materialized on disk.

Crash Recovery
Dealing Gracefully with Failures

Transaction Processing

A transaction is an atomic unit of work in a DBMS

Example: transfer 100 Euro from bank account A to bank account B

Must satisfy the ACID properties:

- Atomic
- Consistent
- Isolated
- Durable

Transaction processing consists of two parts: Crash recovery and Concurrency control

Crash recovery

Is responsible for:

- Atomicity: transactions that are unexpectedly aborted (e.g., due to a system crash) are rolled back and optionally re-executed
- Consistency: by means of atomicity
- Durability: once a transaction is committed its data is persistent through archiving and logging

Several approaches:

- Undo logging
- Redo logging
- Undo/redo logging

See book chapter 17

Concurrency Control
Ensuring Isolation

Concurrency control

Concurrency

To increase throughput and response time, a DBMS will execute multiple transactions at the same time.

Concurrency control ensures that transactions have the same effect as if they were executed in isolation

Concurrency control

Problem: WR conflict

T_1	T_2
`READ(A,s)`	
`s -= 100`	
`WRITE(A,s)`	
	`READ(A,t)`
	`t *= 1.06`
	`WRITE(A,t)`
	`READ(B,t)`
	`t *= 1.06`
	`WRITE(B,t)`
`READ(B,s)`	
`s += 100`	
`WRITE(B,s)`	

Concurrency control

Problem: WW conflict

T_1	T_2
`s = 100`	
`WRITE(A,s)`	
	`t = 200`
	`WRITE(A,t)`
	`t = 200`
	`WRITE(B,t)`
`s = 100`	
`WRITE(B,s)`	

Concurrency control

Definitions

- An action is an expression of the form $r(X)$ or $w(X)$
- A transaction is a sequence of actions.

  $r(A), r(B), w(A), w(B)$

  We abstract away from the actual values read or written.
- A schedule is a sequence of actions belonging to multiple transactions. Subscripts indicate to which transaction an action belongs.

  $r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)$
- A serial schedule is a schedule in which transactions are not executed concurrently. In a serial schedule the actions hence occur grouped per transaction.

  $r_2(A), w_2(A), r_2(B), w_2(B), r_1(A), w_1(A), r_1(B), w_1(B)$

Concurrency control

Serializability

A schedule is called serializable if there exists an equivalent serial schedule.

Example

The following schedules are equivalent:

$$S_1 := r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)$$
$$S_2 := r_1(A), w_1(A), r_1(B), w_1(B), r_2(A), w_2(A), r_2(B), w_2(B)$$

Hence S_1 is serializable.

Concurrency control

Conflict-serializability

- Two actions in a schedule are in conflict if:
  1. they belong to the same transaction; or
  2. act upon the same element, and one of them is a write.

  $$r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)$$

- A schedule is conflict-serializable if we can obtain a serial schedule by (repeatedly) swapping non-conflicting actions.

Example

We can obtain S_2 by swapping only non-conflicting actions from S_1:

$$S_1 := r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)$$
$$S_2 := r_1(A), w_1(A), r_1(B), w_1(B), r_2(A), w_2(A), r_2(B), w_2(B)$$

Consequently S_1 is conflict-serializable.

Concurrency control

Clearly, conflict-serializability implies serializability

The converse is not true

S_1 is equivalent to S_2, but S_2 cannot be obtained from S_1 by conflict-free swapping:

$$S_1 := w_1(Y), w_2(Y), w_2(X), w_1(X), w_3(X)$$
$$S_2 := w_1(Y); w_1(X); w_2(Y); w_2(X); w_3(X)$$

Hence S_1 is not conflict-serializable, but it is serializable.

In practice, a DBMS will only allow conflict-serializable schedules

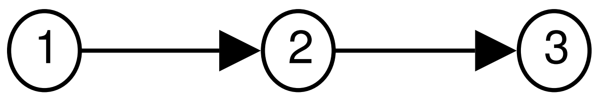
Concurrency control

A simple algorithm to check conflict-serializability

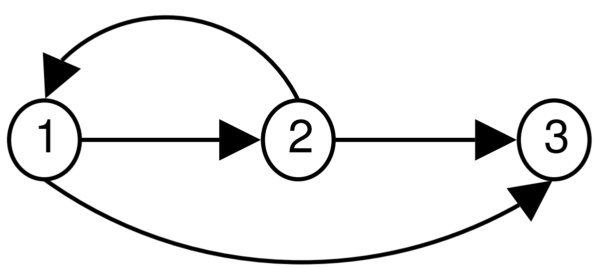
- Construct the precedence graph
- Check whether this graphs contains cycles. If so, output "no", otherwise output "yes"

Example

$$S_1 := r_2(A), r_1(B), w_2(A), r_3(A), w_1(B), w_3(A), r_2(B), w_2(B)$$



$$S_2 := w_1(Y), w_2(Y), w_2(X), w_1(X), w_3(X)$$



Concurrency control

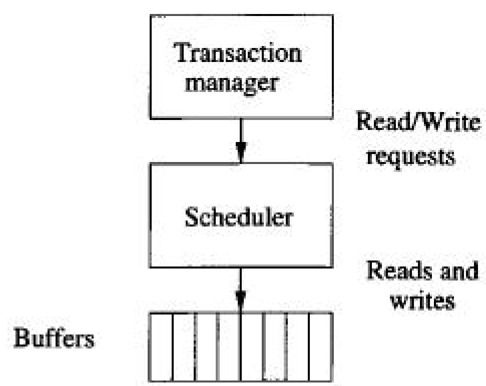
Why does this work?

- If there exists a cycle $T_1 \to T_2 \to \cdots \to T_n \to T_1$ in the dependency graph then we there are actions from T_1 that (1) follow actions from T_n and (2) cannot be moved before the start of T_n by means of conflict-free swapping. Conversely, there are also actions of T_n that follow actions of T_1 and that cannot be moved before T_{n-1} by means of conflict-free swapping. As a consequence, we can never obtain a serial schedule by means of conflict-free swapping (in a serial schedule all actions of T_1 must occur together).
- If there is no cycle in the dependency graph then we can obtain an equivalent serial schedule by topologically sorting the dependency graph. Illustration on the blackboard.
- See Section 18.2.3 in the book

Concurrency control

The scheduler in a DBMS

- It is the taks of the scheduler in a DBMS to create, given a number of transactions, a (conflict-)serializable schedule to be executed.
- New transactions arrive continuously, however, and the scheduler never fully knows the transactions (e.g., because the transactions are large and require a lot of time to run)
- The scheduler hence needs to construct its schedule dynamically, by allowing certain read and write requests; blocking others; and restarting transactions when necessary



Concurrency control

Multiple kinds of schedulers:

- Based on locking
- Based on timestamping
- Based on validation

Concurrency control

Lock-based schedulers

- Add actions of the form $l(X)$ and $u(X)$ to schedules.
- Before an item can be read or written, a transaction must have a lock.
- If transaction i requests a lock that is already taken by another transaction j, the scheduler will pause the execution of i until j releases the lock. It is in particular impossible for two transaction to possess a lock on the same item at the same time.

Concurrency control

Example:

T_1	T_2
$l_1(A), r_1(A)$	
$w_1(A), l_1(B)$	
$u_1(A)$	
	$l_2(A), r_2(A)$
	$w_2(A)$
	$l_2(B)$ denied
$r_1(B), w_1(B)$	
$u_1(B)$	
	$l_2(B), u_2(A)$
	$r_2(B), w_2(B)$
	$u_2(B)$

Concurrency control

Example:

$$l_1(A), r_1(A), w_1(A), u_1(A), l_2(A), r_2(A), w_2(A), u_2(A),$$
$$l_2(B), r_2(B), w_2(B), u_2(B), l_1(B), r_1(B), w_1(B), u_1(B)$$

Question: is this conflict-serializable?

Concurrency control

Two-phase locking

In order to always obtain a conflict-serializable schedule using locks, we require that in each transaction all lock requests precede all unlock requests.

Why is this sufficient to guarantee conflict-serializability?

Illustration on the blackboard. See Section 18.3.3 in book.

Concurrency control

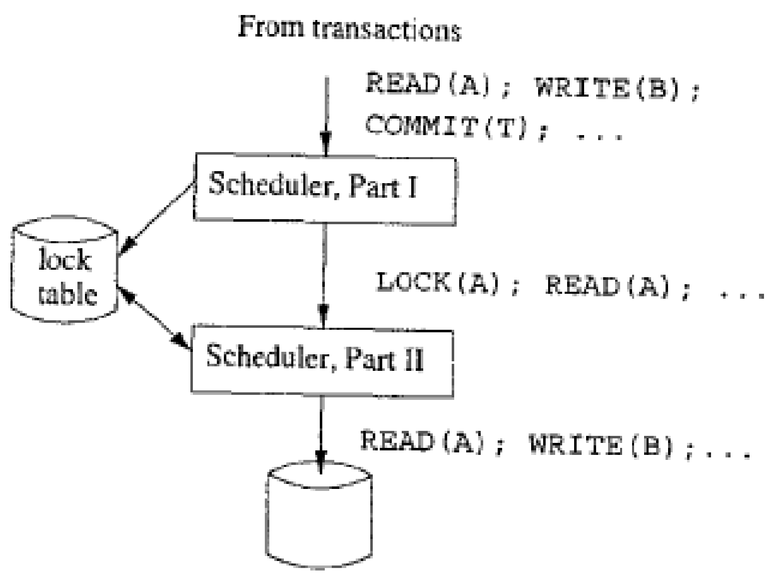
Observe:

- It is harmless for multiple transactions to read the same item at the same time.

  → shared and exclusive locks. See Section 18.4 in book.

- In practice transactions will only make read and write requests. They do not make lock and unlock requests. It is the task of the scheduler to add the latter to the schedule

  → see Section 18.5 in book



Concurrency control

Schedulers based on timestamping

- Are optimistic schedulers
- Assume that we execute transactions T_1, T_2, and T_3 where T_1 was started first, T_2 second, and T_3 third. A timestamping scheduler allows arbitrary reorderings of actions from these transactions, but checks at appropriate times if the reordering used are equivalent to the serial schedule T_1, T_2, T_3. If not, certain transactions are aborted and restarted.

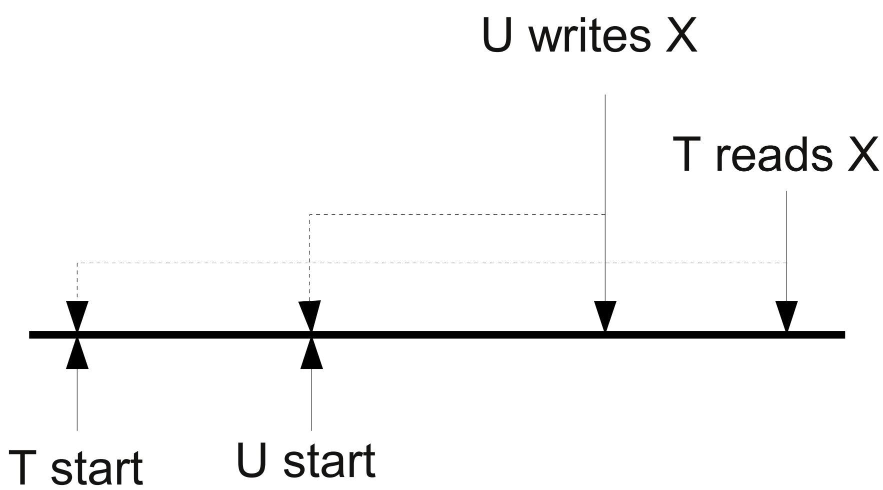
Concurrency control

How does it work?

- Every transaction T receives a timestamp $\mathrm{TS}(T)$ upon creation. This can just be a counter that is incremented for each new transaction.
- To each item X we associate two timestamps $\mathrm{RT}(X)$ and $\mathrm{WT}(X)$, and a boolean $\mathrm{C}(X)$.
  - $\mathrm{RT}(X)$ is the highest timestamp of a transaction that has read X
  - $\mathrm{WT}(X)$ is the highest timestamp of a transaction that has written X
  - $\mathrm{C}(X)$ is true if, and only if, the most recent transaction to write X has already committed.

Concurrency control

Unrealizable behavior that we want to avoid (1/4)

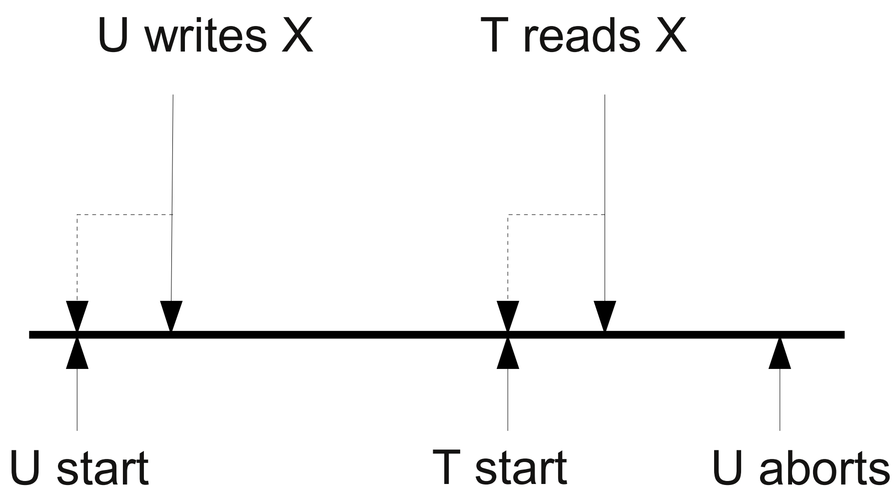


Hence

A read request $r_T(X)$ should only be granted if $\mathrm{TS}(T) \geq \mathrm{WT}(X)$.

Concurrency control

Unrealizable behavior that we want to avoid (2/4)



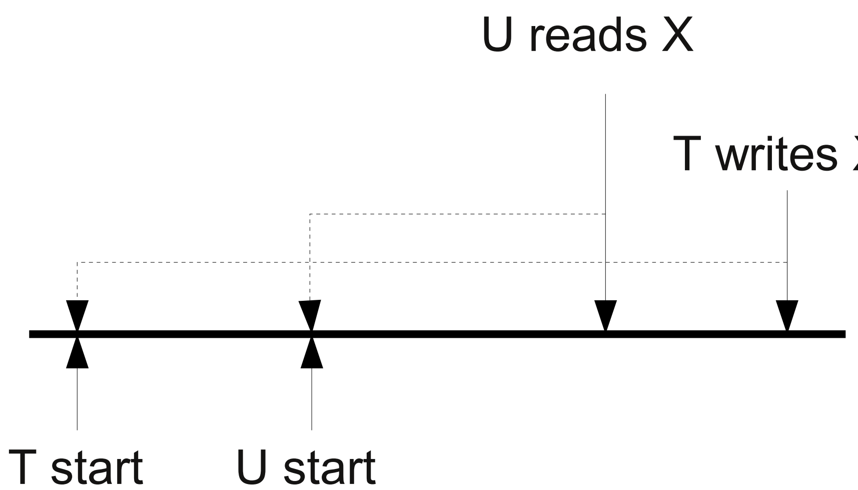
Hence

Read to X should be delayed until the transaction with timestamp $\mathrm{WT}(X)$ commits (i.e., $\mathrm{C}(X)$ becomes true).

Concurrency control

Unrealizable behavior that we want to avoid (3/4)

Suppose $\mathrm{TS}(U) \geq \mathrm{WT}(X)$ at the time when U requests $r_U(X)$.

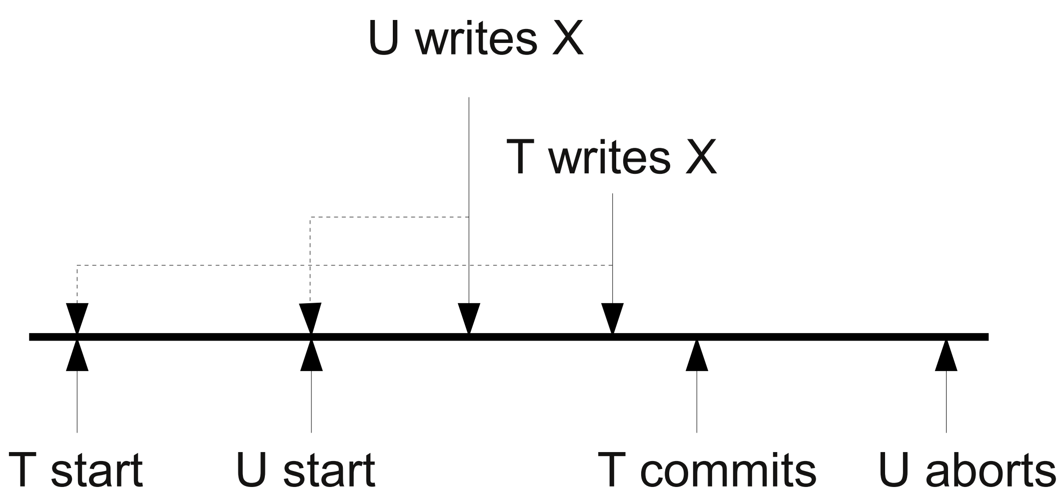


Hence

A write request $w_T(X)$ should only be granted if $\mathrm{TS}(T) \geq \mathrm{RT}(X)$

Concurrency control

Unrealizable behavior that we want to avoid (4/4)



Hence

Request $w_T(X)$ is realizable if $\mathrm{TS}(T) \geq \mathrm{RT}(X)$ and $\mathrm{TS}(T) < \mathrm{WT}(X)$ **BUT**:

- if $\mathrm{C}(X)$ is false then T must be delayed until the transaction with timestamp $\mathrm{WT}(X)$ commits (i.e. $\mathrm{C}(X)$ becomes true)
- if $\mathrm{C}(X)$ is true then the write can be ignored

Concurrency control

How does it work: conclusion

- Every transaction receives a timestamp upon creation. This can just be a counter that is incremented for each new transaction.
- To each item X we associate two timestamps $\mathrm{RT}(X)$ and $\mathrm{WT}(X)$, and a boolean $\mathrm{C}(X)$.
- A transaction with timestamp t is allowed to read item X if $t \geq \mathrm{WT}(X)$. If $\mathrm{C}(X)$ is false then the execution is paused until $\mathrm{C}(X)$ becomes true or the transaction that has last written X aborts. If $t < \mathrm{WT}(X)$ then the transaction is aborted and restarted with a larger timestamp.
- A transaction with timestamp t is allowed to write item X if $\mathrm{RT}(X) \leq t$ and $\mathrm{WT}(X) \leq t$. If $t < \mathrm{RT}(X)$ then the transaction is aborted and restarted with a larger timestamp. If $\mathrm{RT}(X) \leq t < \mathrm{WT}(X)$ and $\mathrm{C}(X)$ is true then we keep the current value of X. Otherwise the execution is paused until $\mathrm{C}(X)$ becomes true, or until the transaction that last wrote X aborts.

Concurrency control

Locking versus timestamping

- Locking is very efficient when we have many transactions that both read and write. In that case, timestamping will need to abort and restart many transactions.
- Timestamping is very efficient when we have many transactions that make only read requests. In that case, many transactions would have to wait for locks when using a lock-based scheduler, while they can immediately proceed with timestamping-based schedulers.

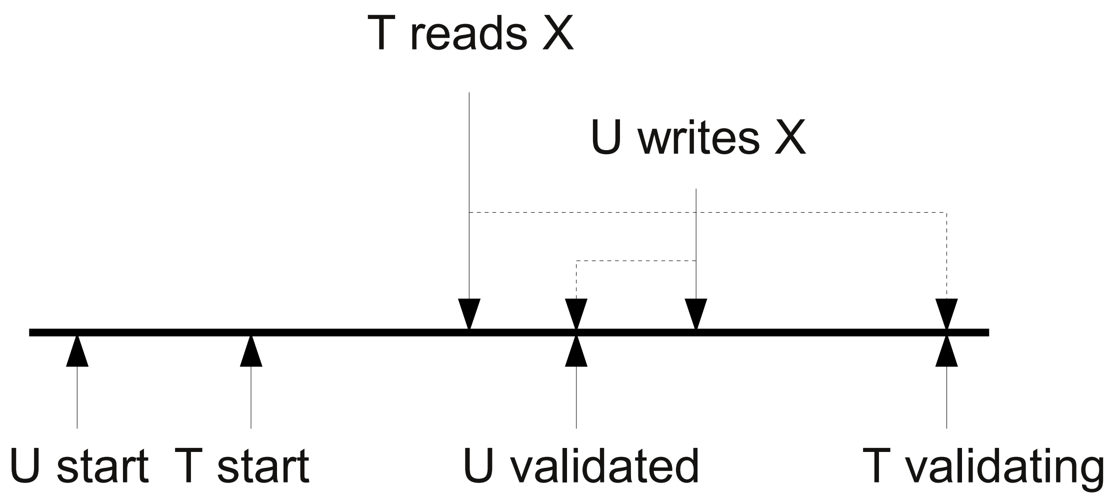
Concurrency control

Schedulers based on validation

- Are optimistic
- The scheduler records, for every transaction T, the set $\mathrm{RS}(T)$ of items read by T, and the set $\mathrm{WS}(T)$ of items written by T.
- Transactions are executed in three phases. In the first phase a transaction reads all items in $\mathrm{RS}(T)$. In the second phase, the scheduler validates the transaction based on $\mathrm{RS}(T)$ and $\mathrm{WS}(T)$. If validation fails, the transaction is aborted and restarted. In the third phase the transaction writes all items in $\mathrm{WS}(T)$.
- The goal is again to obtain a schedule that is equivalent with the serial transaction schedule that orders transactions by their starting time.

Concurrency control

Unrealizable behavior that we want to avoid (1/2)

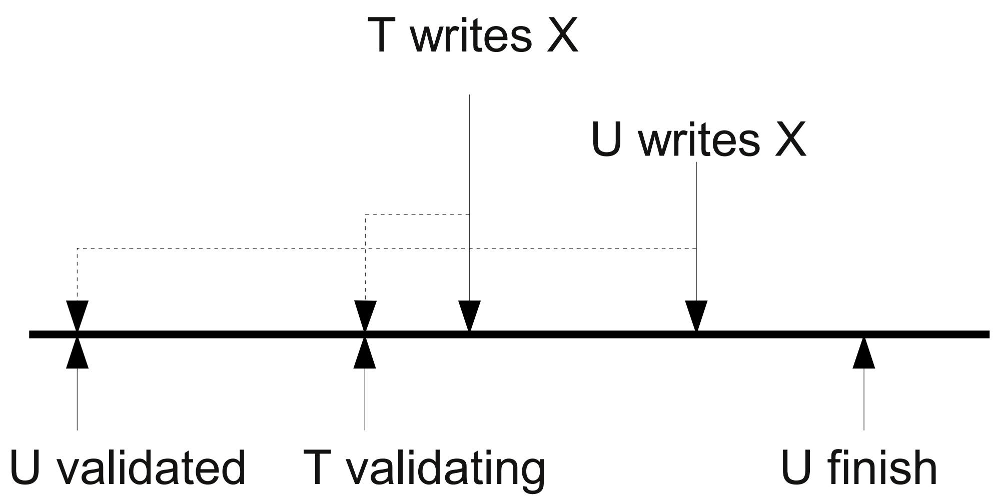


Hence

- Record, for every transaction V, the time $\mathrm{START}(V)$, $\mathrm{VAL}(V)$, and $\mathrm{FIN}(V)$ at which V starts, validates, and finishes, respectively.
- T can only successfully validate if $\mathrm{RS}(T) \cap \mathrm{WS}(U) = \emptyset$ for any previously validated transaction U that was not yet finished when T started, i.e., $\mathrm{FIN}(U) > \mathrm{START}(T)$.

Concurrency control

Unrealizable behavior that we want to avoid (2/2)



Hence

T can only successfully validate if $\mathrm{WS}(T) \cap \mathrm{WS}(U) = \emptyset$ for every previously validated U that did not finish before T validated, i.e., $\mathrm{FIN}(U) > \mathrm{VAL}(T)$.

Concurrency control

How does the scheduler validate?

A transaction T passes validation if:

1. $\mathrm{RS}(T) \cap \mathrm{WS}(U) = \emptyset$ for every transaction U that has already been validated, but was not finished when T started.
2. $\mathrm{WS}(T) \cap \mathrm{WS}(U) = \emptyset$ for every transaction U that has already been validated, but is currently not yet finished.

If T does not pass validation, it is aborted and restarted.

More about transaction management

Interaction between crash recovery and concurrency control

- Crash recovery: recover from system errors by means of logging
- Concurrency control: prevent non-serializable schedules
- Combination?

More about transaction management

Dirty reads

T_1	T_2	A	B
$l_1(A); r_1(A);$		25	25
$A := A + 100;$			
$w_1(A); l_1(B); u_1(A);$		125	
	$l_2(A); r_2(A);$		
	$A := A * 2;$		
	$w_2(A);$	250	
	Commit		
$r_1(B);$			
Crash			

- Recovery problem: T_2 has committed, and can hence not be rolled back. T_1, on the other hand, requires a rollback. But T_2 depends on T_1!

More about transaction management

Dirty reads

T_1	T_2	A	B
$l_1(A); r_1(A);$		25	25
$A := A + 100;$			
$w_1(A); l_1(B); u_1(A);$		125	
	$l_2(A); r_2(A);$		
	$A := A * 2;$		
	$w_2(A);$	250	
	$l_2(B)$ **Denied**		
$r_1(B);$			
Abort; $u_1(B);$			
	$l_2(B); u_2(A); r_2(B);$		
	$B := B * 2;$		
	$w_2(B); u_2(B);$		50

- Implies cascading rollbacks in lock-based schedulers.

More about transaction management

Recoverable schedules

A schedule is called recoverable when every transaction in the schedule commits only when every other transaction from which it has read data, have already committed.

Example

- Recoverable and serial: $S_1 = w_1(A); w_1(B); w_2(A); r_2(B); c_1; c_2;$
- Recoverable, but not serializable: $S_2 = w_2(A); w_1(B); w_1(A); r_2(B); c_1; c_2;$
- Not recoverable, but serializable: $S_3 = w_1(A); w_1(B); w_2(A); r_2(B); c_2; c_1;$

Notice that

Recoverable schedules like S_1 may still require a cascading rollback!

More about transaction management

Avoid Cascading Rollback (ACR) schedules

A schedule is said to avoid cascading rollbacks if transactions in the schedule only read data from transaction that have already committed. In other words: the transaction can never ready "dirty data".

Example

- ACR and serial: $S_4 = w_1(A); w_1(B); w_2(A); c_1; r_2(B); c_2;$

Observe:

Every ACR schedule is recoverable.

More about transaction management

Strict schedules

A lock-based schedule is strict when every transaction only releases its exclusive locks when it has committed or aborted, and the commit or abort log record has been written to disk.

Observe:

- Every strict schedule is ACR.
- Every strict schedule is serializable.

Simplification

We need not wait until the commit or abort log record has been written to disk, provided that we are guaranteed that log records are written in the same order as they are created (group commit).