SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

Database Systems
WS 08/09

Prof. Dr. Jens Dittrich

Chair of Information Systems Group
http://infosys.cs.uni-saarland.de

Topics (3/6)

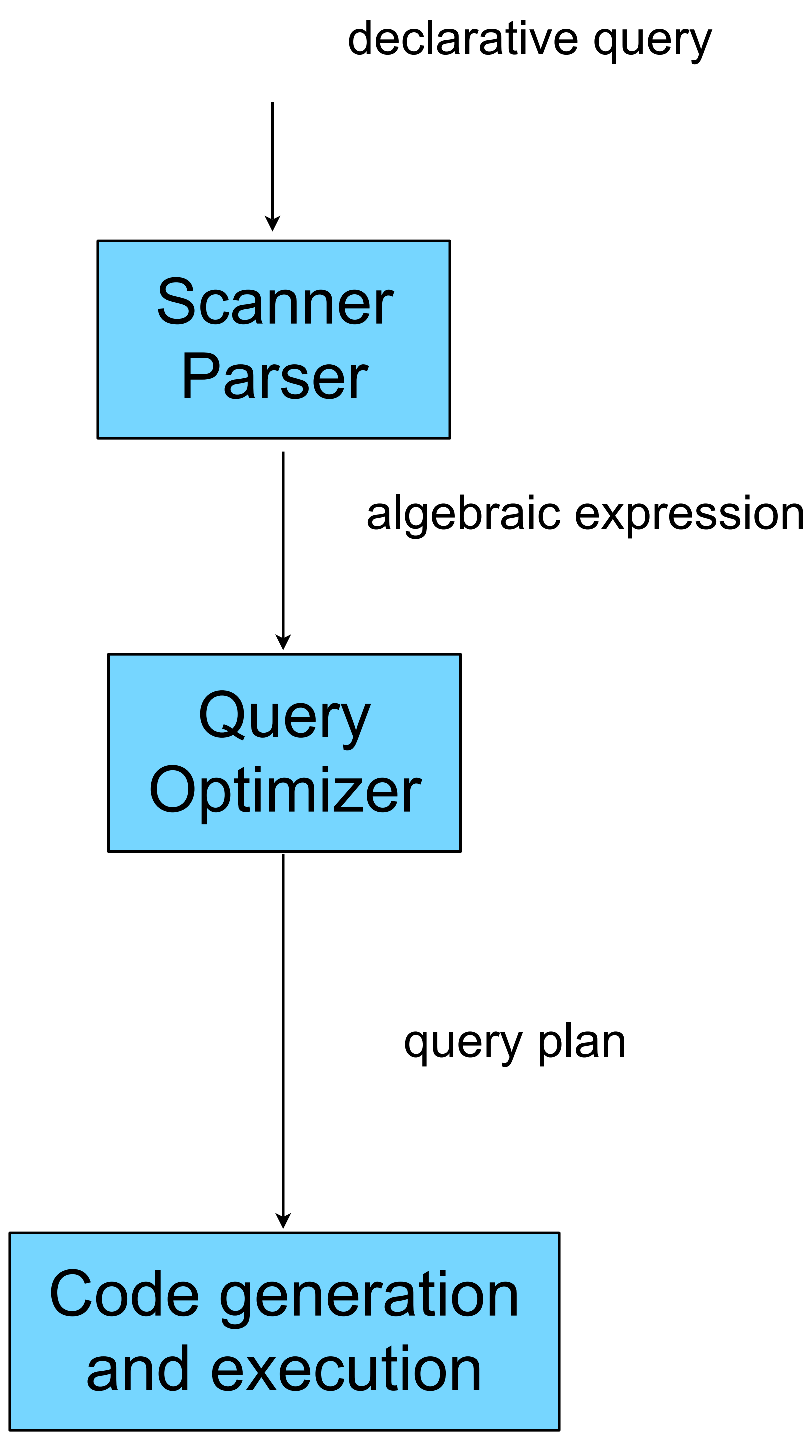
- operator models
  - push-model
  - pull-model
- operator implementations
  - general idea
  - join algorithms for relational and multidimensional data
  - other operators
- query processing
  - scanning & “naive plans“
  - canonical plan computation

Operator Models.

Operator Models.

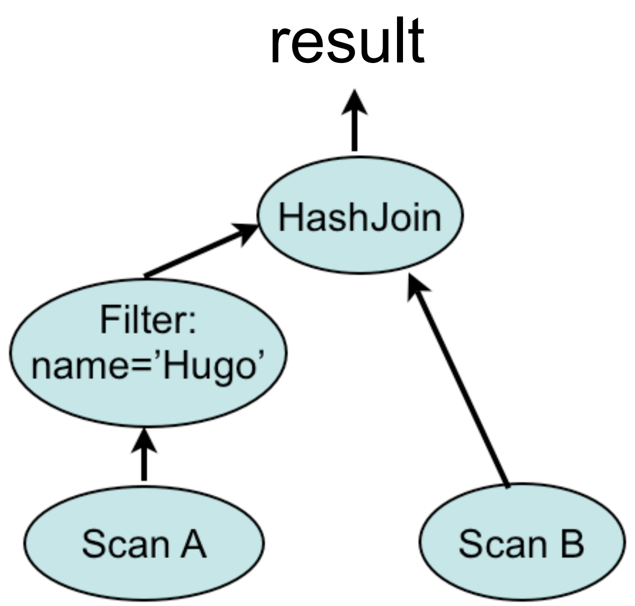
Introduction.

Motivation & Overview



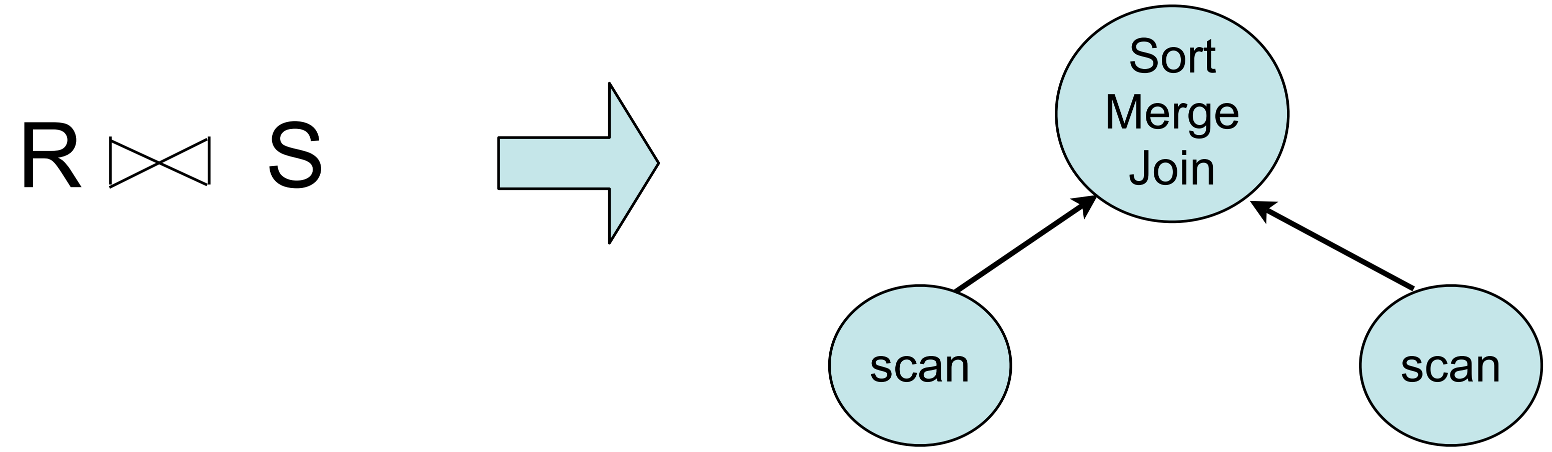

```
SELECT title
FROM A,C
WHERE A.name ='Hugo' AND A.id = C.dz
```

$$\prod_{\mathtt{title}} \left(\sigma_{\mathtt{A.name='Hugo'\ and\ A.id=B.dz}} \left(\mathtt{A \times B} \right) \right)$$

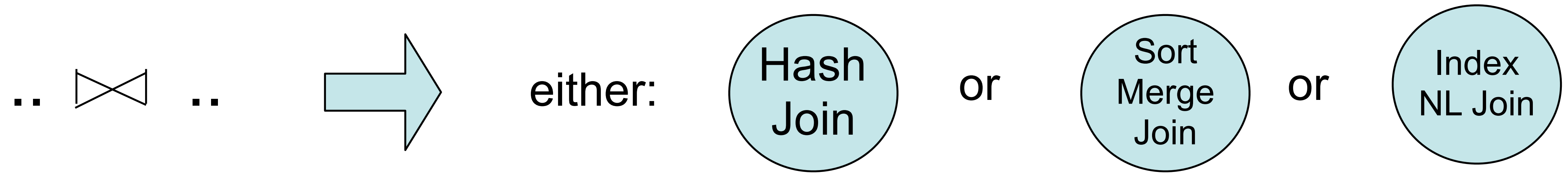


Logical and Physical Operators

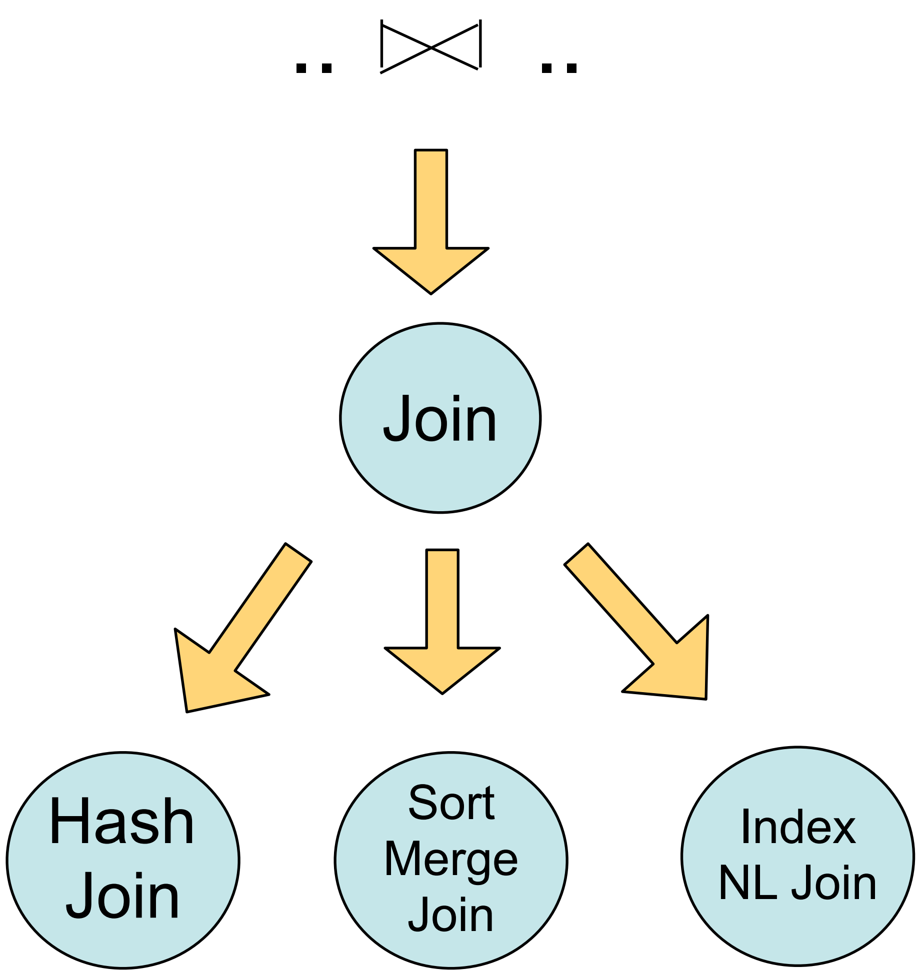
- Logical operators are the atoms of a query plan
- Every logical operator computes a function f()
- Every logical operator will be mapped to one or multiple physical operators (implementations of f())



- there may be different implementations (physical operators) for the same function (logical operator)



Operators: Logical vs. Physical



algebraic expression

logical operator

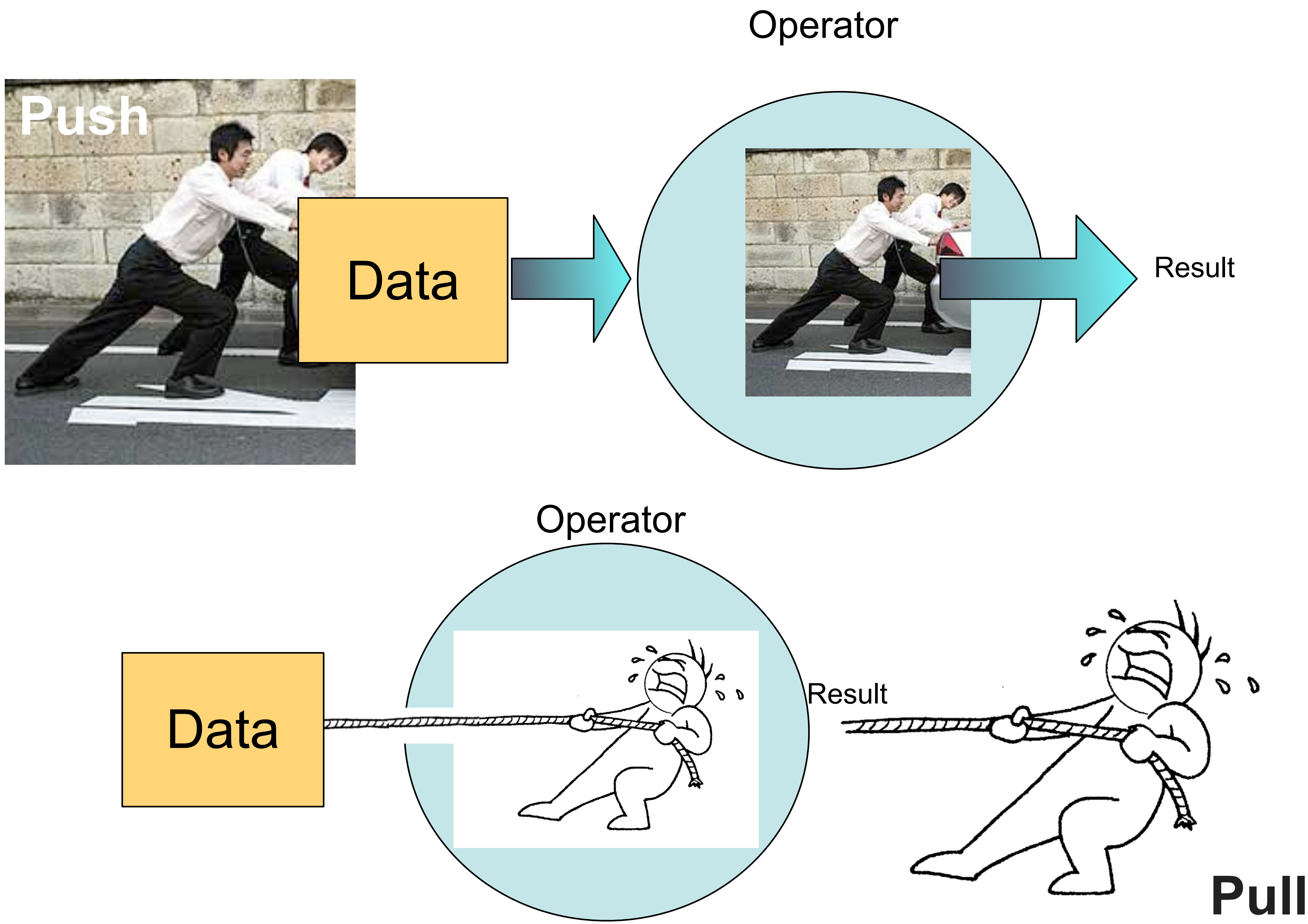
physical operators

Today: physical operators
(we are proceeding bottom-up)

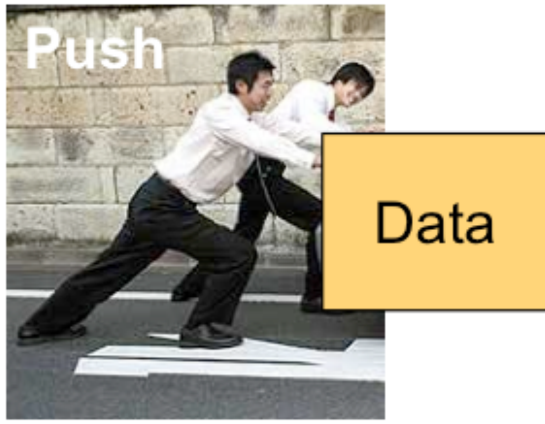
Operator Models.

Push versus Pull.

Push vs. Pull



Push (Stream-Model)



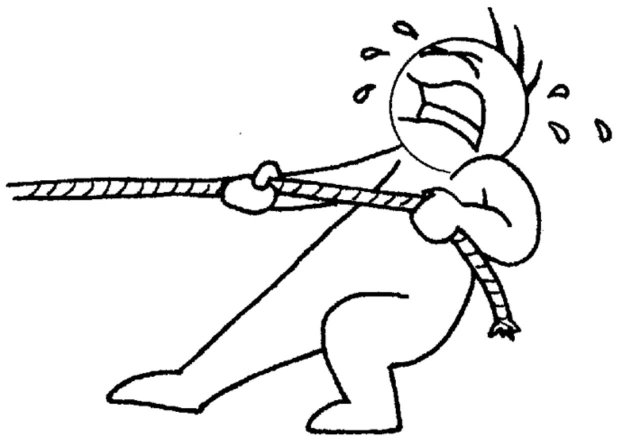
- data sources generate data and send them to the next operator
- data processing is triggered by the data sources
- data processing is **data driven.**
- Example
  - temperature sensors send measurements to a local center
  - local center aggregates measurements and sends it to a center responsible for a bigger area, etc.
- Advantage:
  - every data items is processed immediately after it was generated
- Disadvantage:
  - receiver may not easily suspend or stop data generation

PushOperators: Implementation

```java
package core.tools;


public interface PushOperator {

    /**
     * Passes the next element to the consumer.
     *
     * @param element
     */
    public void pass(int element);

    /**
     * Announces end of stream to the consumer.
     */
    public void finished();

}
```

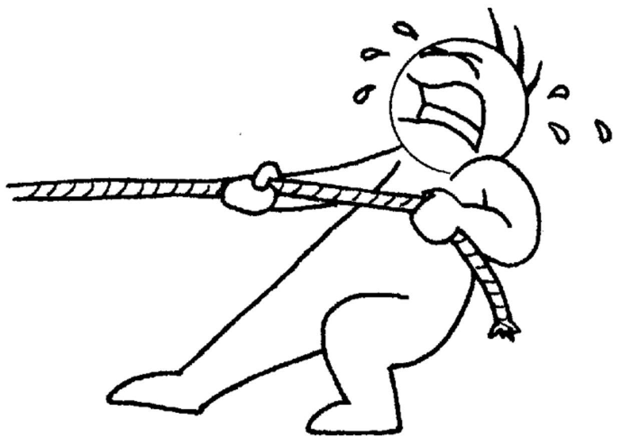
- Advantage: avoids to have separate thread for each operator
- state-of-the-art for streaming systems
- e.g., XXL PIPES library
- also useful for other situations where pull-operators are hard to use

Pull (Iterator-Model)



- user requests next result **from** the data source
- data processing is triggered by the user
- data processing is **user driven**
- Example:
  - "Computer, show me the next pizzeria."
  - pause.
  - " Computer, show me the second-next pizzeria."
  - pause.
- advantage:
  - computer does not compute results that were not requested (i.e., too many results)
- disadvantage:
  - extra effort for propagating user calls back to the data sources

Pull: Implementation

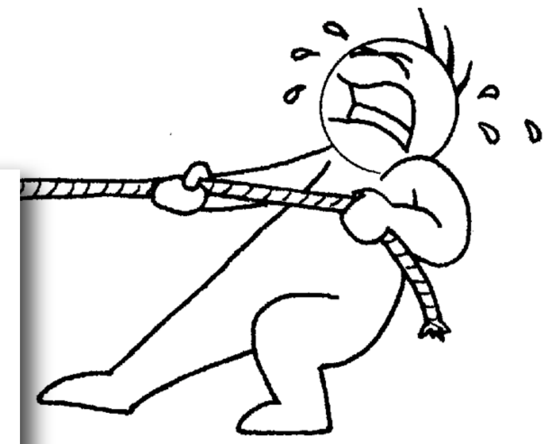


- implementation based on operator-Interface
- void **open**():
  initialized the operator
- Object **next**():
  returns the next element and removes it from the operator
- void **close**()
  closes the operator

This interface is also called **ONC**-interface.

- Literature: Goetz Graefe: Volcano - An Extensible and Parallel Query Evaluation System. IEEE Trans. Knowl. Data Eng. 6(1): 120-135 (1994)

java.util.Enumeration (Java 1.0)



java.util

Interface Enumeration

All Known Subinterfaces:
NamingEnumeration

All Known Implementing Classes:
StringTokenizer

---

public interface **Enumeration**

An object that implements the Enumeration interface generates a series of elements, one at a time. Successive calls to the `nextElement` method return successive elements of the series.

For example, to print all elements of a vector *v*:

```
for (Enumeration e = v.elements() ; e.hasMoreElements() ;) {
    System.out.println(e.nextElement());

}
```

Methods are provided to enumerate through the elements of a vector, the keys of a hashtable, and the values in a hashtable. Enumerations are also used to specify the input streams to a `SequenceInputStream`.

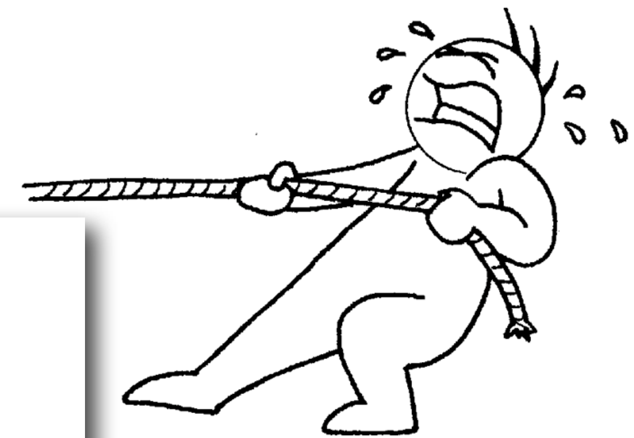
NOTE: The functionality of this interface is duplicated by the Iterator interface. In addition, Iterator adds an optional remove operation, and has shorter method names. New implementations should consider using Iterator in preference to Enumeration.

Since:
JDK1.0

See Also:
`Iterator`, `SequenceInputStream`, `nextElement()`, `Hashtable`, `Hashtable.elements()`, `Hashtable.keys()`, `Vector`, `Vector.elements()`

Method Summary	
`boolean`	`hasMoreElements()` Tests if this enumeration contains more elements.
`Object`	`nextElement()` Returns the next element of this enumeration if this enumeration object has at least one more element to provide.

java.util.Iterator (Java 1.2)



java.util

Interface Iterator

All Known Subinterfaces:
ListIterator

All Known Implementing Classes:
BeanContextSupport.BCSIterator

---

public interface **Iterator**

An iterator over a collection. Iterator takes the place of Enumeration in the Java collections framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the Java Collections Framework.

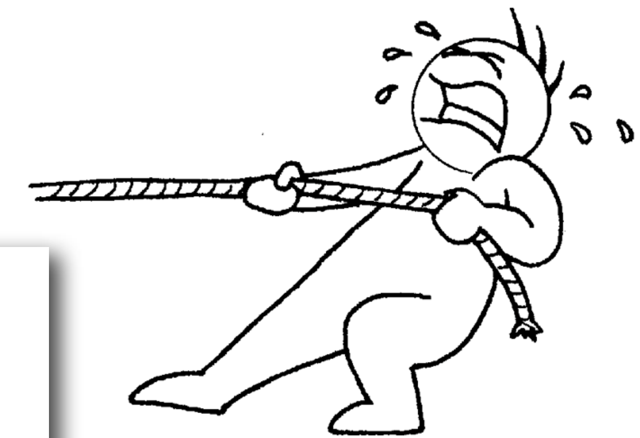
Since:
1.2

See Also:
`Collection`, `ListIterator`, `Enumeration`

Method Summary

`boolean`	**`hasNext()`** Returns `true` if the iteration has more elements.
`Object`	**`next()`** Returns the next element in the iteration.
`void`	**`remove()`** Removes from the underlying collection the last element returned by the iterator (optional operation).

java.util.Iterator<E> (Java 1.5)



java.util

Interface Iterator<E>

All Known Subinterfaces:
ListIterator<E>

All Known Implementing Classes:
BeanContextSupport.BCSIterator, Scanner

```
public interface Iterator<E>
```

An iterator over a collection. Iterator takes the place of Enumeration in the Java collections framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

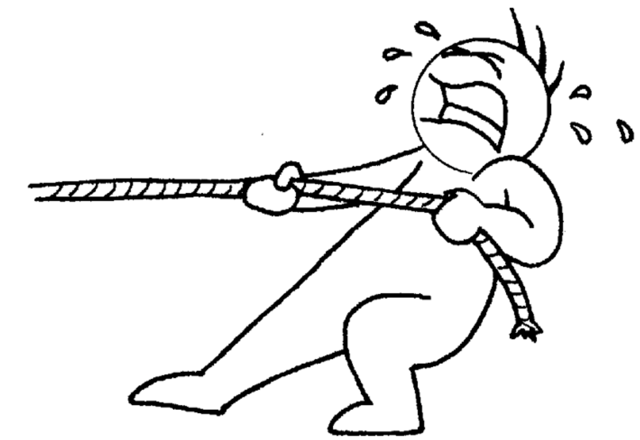
This interface is a member of the Java Collections Framework.

Since:
1.2

See Also:
`Collection`, `ListIterator`, `Enumeration`

Method Summary	
`boolean`	`hasNext()` Returns `true` if the iteration has more elements.
`E`	`next()` Returns the next element in the iteration.
`void`	`remove()` Removes from the underlying collection the last element returned by the iterator (optional operation).

java.util.Iterator



- void **open**():
  not available, implicitly done by the construct
- boolean **hasNext**():
  returns true if the iterator may deliver another element
- Object **next**():
  returns the next element and removes it from the iterator
- void **close**()
  not available, finalize()-method? (a bit risky)

```java
public static void main(String argv[]) throws Exception {

    Object object = new Object() {

        protected void finalize() {
            System.err.println("test");
        }

    };
    System.err.println(object);

}
```

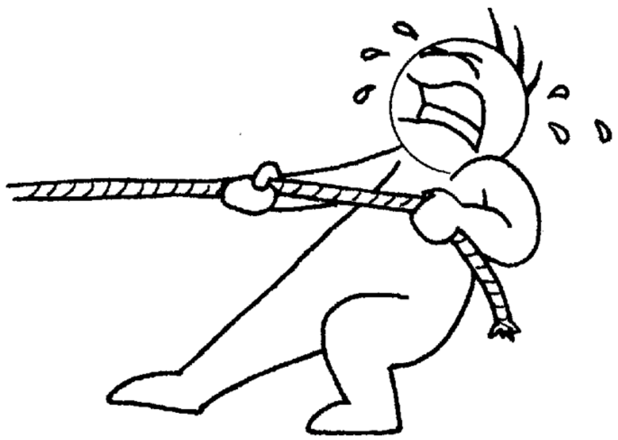
```
Console  Search  Tasks  Problems
<terminated> FolderList [Java Application] /System/Library/
Test$1@6a55fa
```

Operator Implementations.

Operator Implementations.

Examples of Pull Operators.

Iterators: Simple Examples



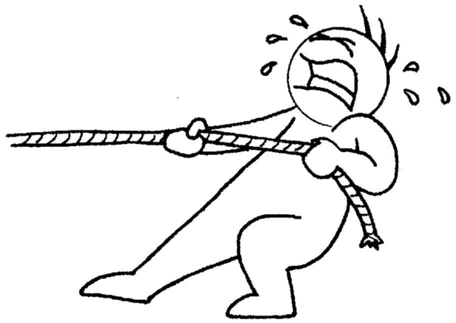
- **Projection {**

```
Iterator input;
public Object next() {
    return project( input.next() );
}}
```

- **Selection {**

```
Iterator input;
public Object next() {
    Object result = null;
    do {
        result = input.next();
    } while ( !P(result) )
    return result;
}}
```

Iterators: Loops



- **put loop into next()? NO!**

```
public Integer next() {
    for(int i=0; i<42; i++){
        return i;
    }
    throw new NoSuchElementException();
}
```

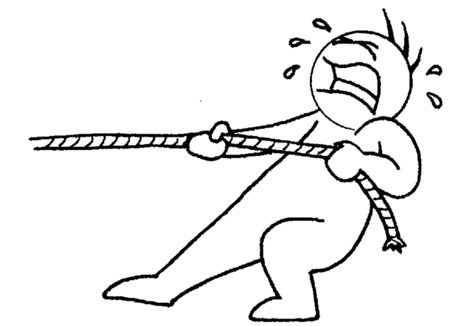
How do I suspend the computation here?

...and continue at the same position later on?

- **Solution: persist the state of the operator**

```
int i=0;  // loop-counter becomes an attribute of the class
public Integer next() {
    while(i<42) {
        Integer result = i;
        i++;
        return result;
    }
    throw new NoSuchElementException();
}
```

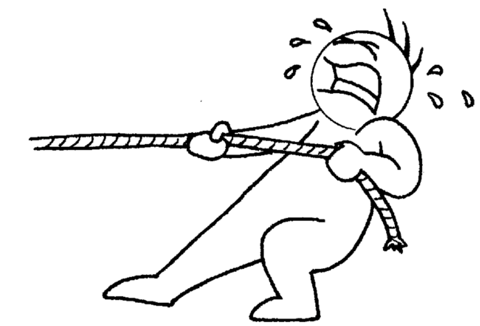
AbstractIterator: computeNext()



```java
public abstract class AbstractIterator implements Iterator {

    protected Object next = null;

    protected Iterator results = null;

    protected boolean hasNext = false;

    protected boolean setNextIterator(Iterator results) {...

    protected boolean setNext(Object result) {...

    protected boolean getNext() {...

    abstract protected boolean computeNext();

    public boolean hasNext() {...

    public Object next() {...

    public void remove() {...

}
```

computeNext() will be called whenever new results have to be computed.
Only this method has to be provided by the developer.

AbstractIterator: setNext(Object)

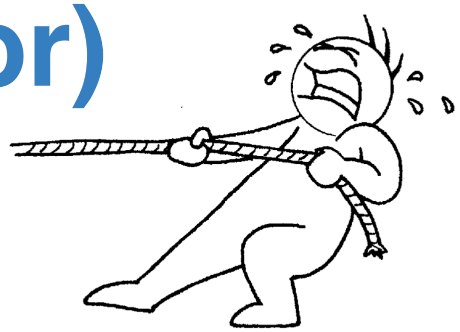


```
public abstract class AbstractIterator implements Iterator {
    protected Object next = null;
    protected Iterator results = null;
    protected boolean hasNext = false;
    protected boolean setNextIterator(Iterator results) {...
    protected boolean setNext(Object result) {...
    protected boolean getNext() {...
    abstract protected boolean computeNext();
    public boolean hasNext() {...
    public Object next() {...
    public void remove() {...
}
```

whenever the developer wants to deliver a result item he writes:

return setNext(instance);

AbstractIterator: setNextIterator(iterator)

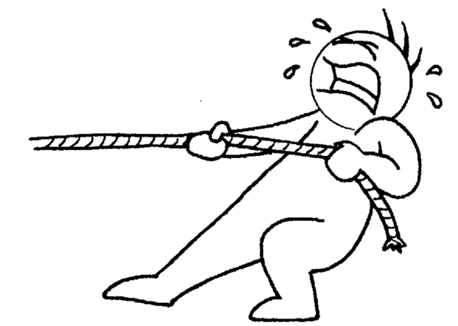


```
public abstract class AbstractIterator implements Iterator {

    protected Object next = null;

    protected Iterator results = null;

    protected boolean hasNext = false;

    protected boolean setNextIterator(Iterator results) {...

    protected boolean setNext(Object result) {...

    protected boolean getNext() {...

    abstract protected boolean computeNext();

    public boolean hasNext() {...

    public Object next() {...

    public void remove() {...

}
```

whenever the developer wants to deliver an iterator of result items he writes:

return setNextIterator(iterator);

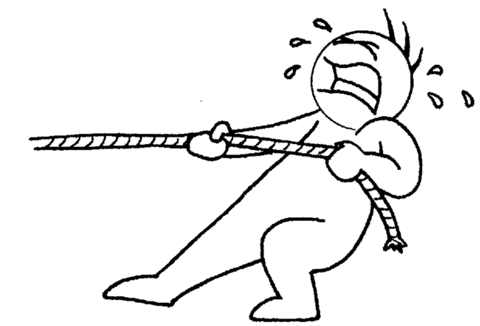
AbstractIterator: Example



```java
import java.util.Iterator;

public class SortBasedDistinct extends AbstractIterator {

    protected Iterator input = null;

    protected Object peek = null;

    public SortBasedDistinct(Iterator input) {..

    protected boolean computeNext() {
        if (this.input.hasNext()) {

            //compute next element <_next> here

            return setNext(_next);
        } else
            return false;
    }
}
```

only a few lines have to be provided in computeNext(). Everything else is done by AbstractIterator!

AbstractIterator as the Return Value of a Method

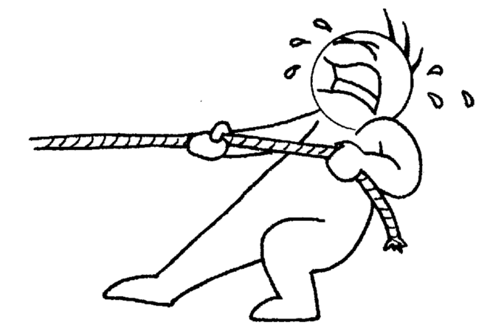


query result has to be returned as an iterator

```java
import java.util.Iterator;

public class QueryProcessor {

    public QueryProcessor() {
    }

    public Iterator query(String queryExpression) throws Exception {
        return new AbstractIterator() {
            protected boolean computeNext() {
                // put your code here
                return false;
            }
        };
    }
}
```

Solution: anonymous class (implicitly non-static!)

Static vs. non-static Inner Classes



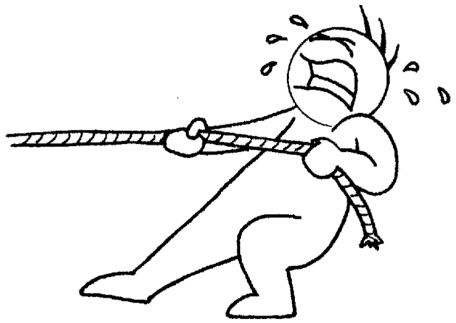
```java
public class QueryProcessor {

    protected List myList = null;

    public static class MyIteratorStatic extends AbstractIterator {

        protected boolean computeNext() {
            myList.get(42);
            return false;
        }

    }

    public class MyIteratorNonStatic extends AbstractIterator {

        protected boolean computeNext() {
            myList.get(42);
            return false;
        }
    }

    public QueryProcessor() {...

    public Iterator query(String queryExpression) throws Exception {
        return new AbstractIterator() {
            protected boolean computeNext() {
                if (myList.size() > 42) {
                    return setNext(myList.get(42));
                }
                return false;
            }
        };
    }
}
```

access **not** allowed!
This **static** inner class does not have a reference to the outside instance of QueryProcessor.

This **non-static** inner class has a reference to the outside instance of QueryProcessor!

Access allowed!
This anonymous class is implicitly **non-static**.

AbstractIterator and Constructors



```java
public Iterator query(String queryExpression) throws Exception {
    return new AbstractIterator() {

        public AbstractIterator() {
            //code to initialize this instance
        }

        protected boolean computeNext() {
            if (myList.size() > 42) {
                return setNext(myList.get(42));
            }
            return false;
        }
    };
}
```

overloading the constructor is not allowed (unfortunately)

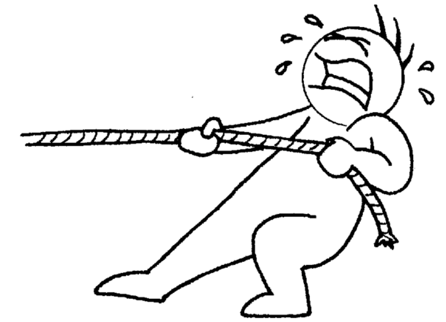
```java
public Iterator query(String queryExpression) throws Exception {
    return new AbstractIterator() {

        {
            //code to initialize this instance
        }

        protected boolean computeNext() {
            if (myList.size() > 42) {
                return setNext(myList.get(42));
            }
            return false;
        }
    };
}
```

Solution:
instance-initializer will be called before the constructor is called

Operator Implementations.

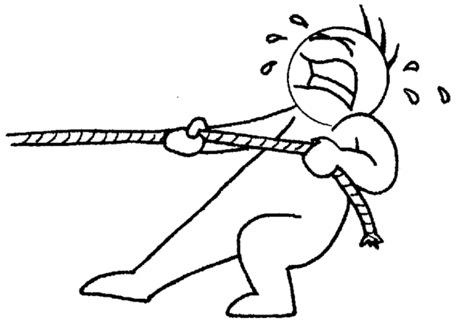
Granularity of Iteration.

Granularity of Iteration



- so far we assumed that only one element will be returned by a call to next().
- However, each call to next() may trigger several cascading calls to underlying iterators
- Drawbacks:
  - single (or few) item processing in operator graph
  - hardly any bulk-operation inside operators
  - cache misses
  - many function calls
- Advantages:
  - may be a good model for an external memory system that tries to touch as little data as possible

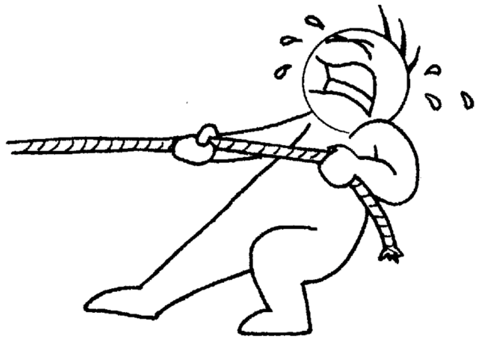
Coarse-Granular Iterators



- better solution: block-wise iteration
- iterate on blocks/groups of data
- For instance: consider 100 tuples at a time
- Advantages:
  - less function calls
  - better cache behavior (locality!)
- Drawbacks:
  - may read unnecessary data from external memory
- Trade-off among

fine-granular iterators	⟷	coarse-granular iterators
many method calls		less method calls
clean/easy design		more complicated design
bad performance		better performance

Operators: Overview



- most DBMS only implement the pull-model (unfortunately).
- important operators:
  - (external) sorting
  - joins
  - grouping and aggregation
  - selection
  - division
  - intersection
- Furthermore
  - full-table scan (FTS)
  - IndexScan

Operator Implementations.

External Sorting.

Motivation for External Sorting

- bulk-loading of a tree index:
  - first: sort data
  - then: build index
- user wants output in some order, e.g., increasing age
- sorting to eliminate duplicates
- sorting as the basis for sort-based join algorithms, e.g.
  - sort-merge-join
  - plane sweep om spatial data

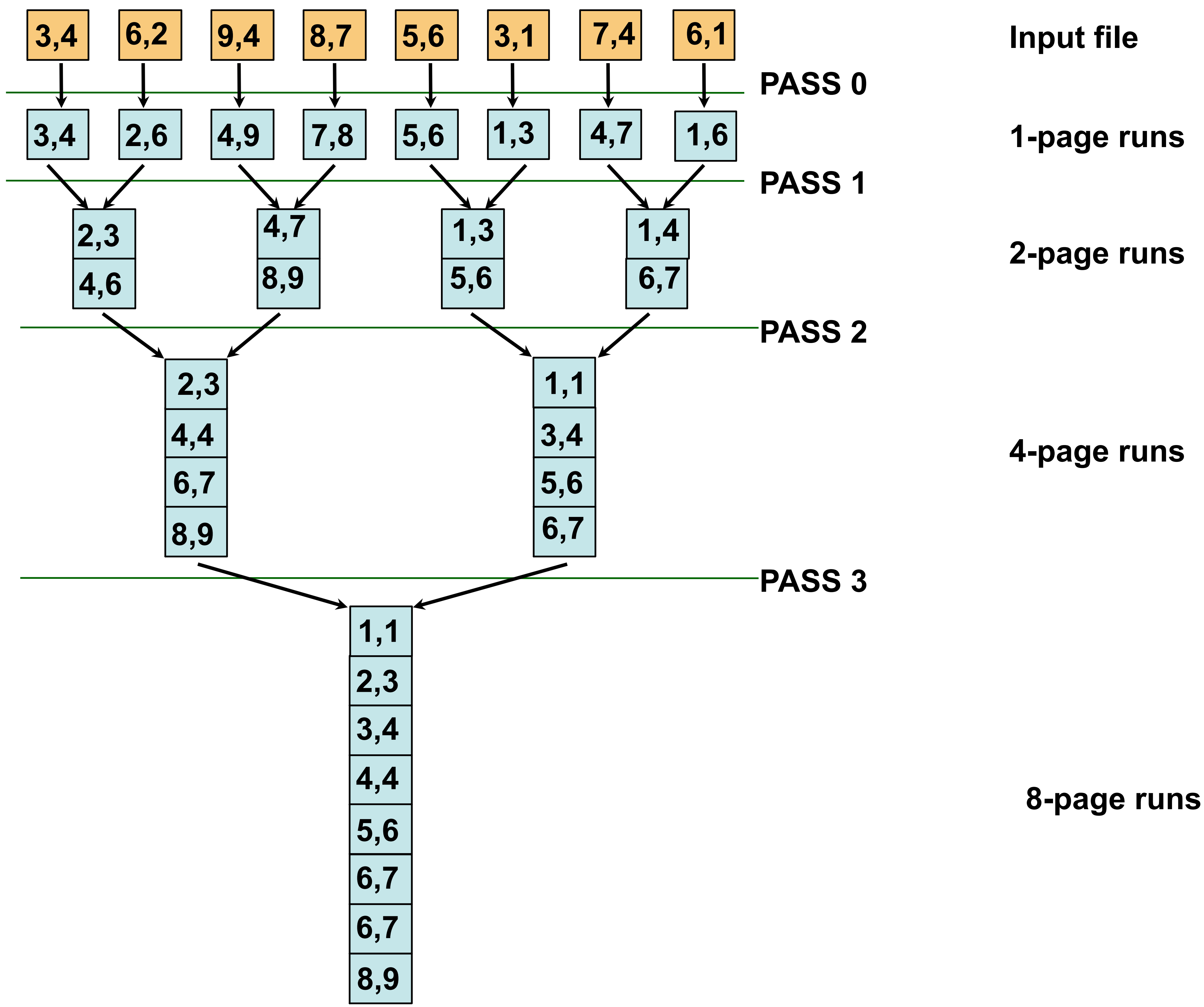
Simple Two-way Merge Sort

- idea of Phase 1:
  - read each page separately
  - sort it in main memory
  - write it back to external memory
  - the data that is written back is called a **run**
  - Note: this requires only **1 page of main memory**
- Idea of Phase 2:
  - consider two input runs
  - read each run into main memory
  - merge pages into a sorted 2-page run
  - Note: this requires only **3 pages of main memory**
  - whenever the output page is full write it to output run and start new page
  - loop until all 1-page runs have been merged to 2-page runs

Simple Two-way Merge Sort

- Idea of Phase 3:
  - consider two input runs of two pages
  - read one page from each run into main memory
  - merge runs into a sorted 4-page run
  - Note: this requires only **3 pages of main memory**
  - whenever the output page is full, write it to output run and start new page
  - whenever the input page is empty, read a new input page from the run (if there is another page)
  - loop until all 2-page runs have been merged to 4-page runs
- perform recursive merging until only a single sorted run is created

Example: Two-Way External Merge Sort



How to Improve this?

- Assume we have m pages in main memory available
- Assume data set of size n pages
- Phase 1: read m pages into main memory and sort them
- Merge Phases: merge m-1 input runs into one output run
- Again: for each input and output run only one page is required
- Note: Last run created in Phase 1 may have less then m pages
- Effects
  - Phase 1: creates less runs: $\lceil n/m \rceil$ instead of n
  - Phase 2: number of merge phases reduced:

$$\lceil \log_{m-1} \lceil n/m \rceil \rceil \text{ instead of } \lceil \log_2 n \rceil$$

External Sorting: Core Algorithm

- parameters:
  - B: number of records per page
  - N: number of records
  - n: = $\lceil N/B \rceil$, number of pages
  - M: available main memory in records
  - m: = $\lceil M/B \rceil$, available main memory in pages
- Problem: N records do not fit into main memory simultaneously
- Algorithm:

  1. Phase: Run-generation

     While R not empty:

     Load M records of R into main memory

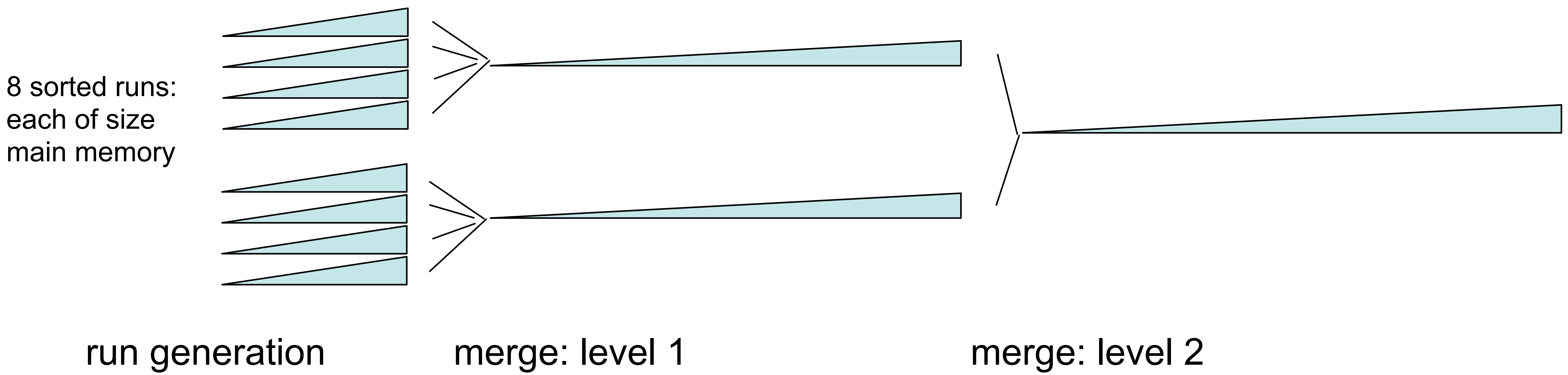
     Sort M records in main memory and write sorted sequence to a temporary file called a **run**

  2. Phase: Recursive merge

     While (number of runs >1):

     Merge F runs from disk into a single bigger run

External Sorting



- F: fan-in of the merge: F= m-1 = O(m)
- number of merge levels: $\lceil \log_F n \rceil$
- I/O-effort for merging: $O(n \log_m n)$
- in total: the asymptotic cost for external sorting are

$$O(n \log_m n)$$

Blocked I/O

- so far we optimized externa algorithm reducing the number of merge-levels
- however: random I/O for m-1 input runs will be considerable
- consider m=100,001
- => a merge creating an output run of size 100,000 pages will on average require 100,000 random I/O-operations...
- therefore
  - we need to reduce the fan-in
  - assign multiple pages for each input and output buffer
  - decreases amount of random I/O
- What is the best value for the fan-in?
  - model the expected I/O cost
  - measure on the specific hardware (calibrate)

Double Buffering

- so far: one or multiple pages fo an input or output
- problem:
  - synchronous requests
  - CPU will wait (idle) for the data to come in
- improvement
  - assign two buffers for each input and output
  - one buffer is currently considered by the CPU
  - the other is filled or emptied by the disk
    (using a separate thread)
- Example for reading:
  - CPU reads data from first buffer
  - in the background hard disk fills second buffer
  - if first buffer is empty switch roles of both buffers

Replacement Selection

- Observation: length of an initial run is equal to the available memory: M
- Is it possible to generate initial runs that are longer than M?
- Yes!
- R = input, R'= output run, h=heapsize
- **Algorithm: Replacement Selection**

```
fill main memory with M records of R
create a heap on M records, h:=M
while heap not empty:
    write top-element of heap r to R'
    let t be the next element of input R
    If t >= r:
        insert t into heap
    Else: (t<r)
        h:=h-1 // decrease heap size
        store t at freed slot in main memory
```

Replacement Selection Example

M=4

time ↓

output run R‘						main memory				input R						
						10	20	30	40	25	73	16	26	33	50	31
					10	20	25	30	40	73	16	26	33	50	31	
				10	20	25	30	40	73	16	26	33	50	31		
			10	20	25	(16)	30	40	73	26	33	50	31			
		10	20	25	30	(16)	(26)	40	73	33	50	31				
	10	20	25	30	40	(16)	(26)	(33)	73	50	31					
10	20	25	30	40	73	(16)	(26)	(33)	(50)	31						
					16	26	31	33	50							

output run contains 6>M elements

- start: main memory contains elements 10, 20, 30, 40
- iteratively the next element t is taken from input R
- if t greater (equal) than the last top-element: insert into heap
- if t smaller than the last top-element: decrease heap size, insert into main memory

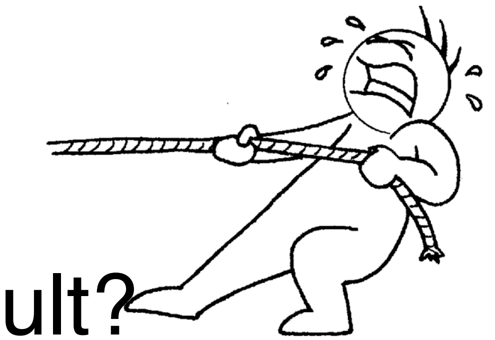
Discussion

- on average replacement selection creates runs of length 2M (!)
- if input data is presorted, runs may even be longer than that
- Advantage: reduces number of merge levels
- Disadvantages: heap-sort not cache-conscious

Improvement: Online Merge

- Idea:
  - the final merge of external sorting may be performed online
  - the final sorted sequence will be passed directly to the next operator
- For instance:
  - consider that the output of the sort is the input to some other operator (e.g., a join)
  - assume that the sorted sequence is not needed otherwise
  - Therefore it does not make sense in this situation to write the entire sorted sequence to a single final run!!
- Advantage
  - we save reading of n pages and writing of n pages
  - in total: 2n I/O-operations are saved (including considerable random I/O)
- Should be used for:
  - joins, aggregation, bulk-loading, index inversion, etc.

Blocking Operators



- how may a sorting operator determine the next result?
- a sorting operator may only decide which element has to be returned by next() **after** it has seen all input elements!
- In other words: the sorting operator first consumes all its input elements before delivering any result item:

it **blocks** the data flow.

Operator Implementations.

Join Algorithms.

Join-Algorithms

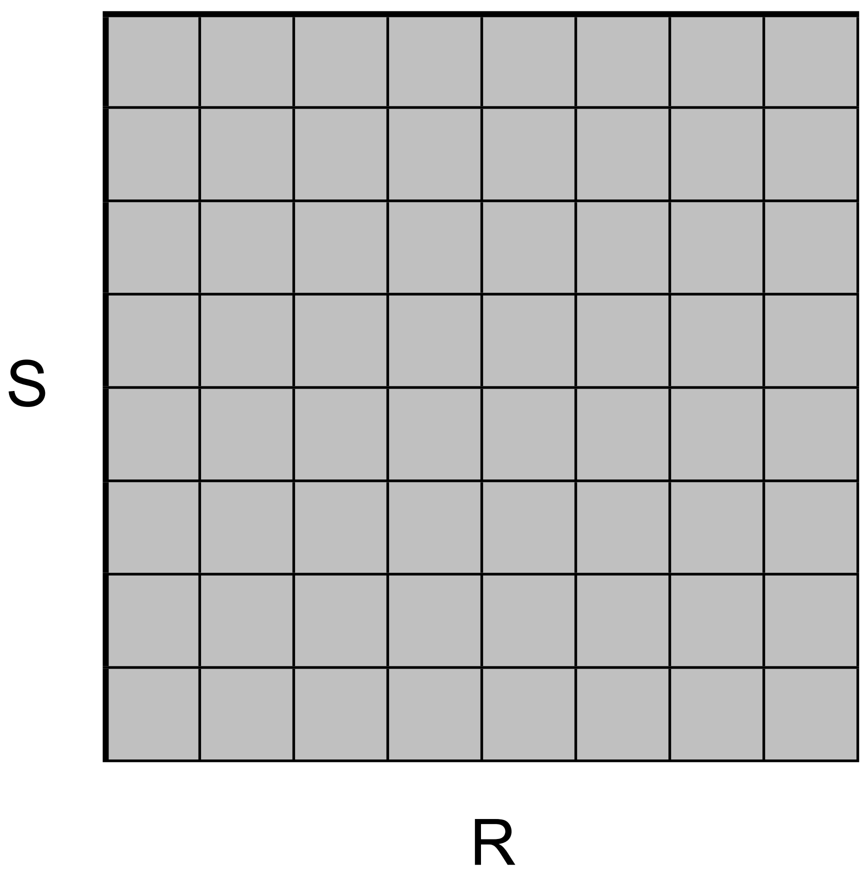
- **Role of joins**
  - most important operation in a DBMS
  - considerable impact on overall performance of a DBMS
  - considerable impact on blocking behavior of queries, i.e., time required to report the first result tuple
  - join techniques very similar to techniques needed to implement other operators
    Examples: intersect, minus, complement
- 3 important classes of algorithms
  - Nested-loops joins
  - Hash joins
  - Sort-merge joins

Simple Nested-Loops Join

- Input:
  - Relation R
  - Relation S
  - Join-predicate P(r,s)
  - result set RES = {}
- Algorithm:

```
ForEach r in R:
    ForEach s in S:
        If P(r,s):
            RES = RES ∪ { (r,s) }
```

- Discussion:
  - |R| × |S| comparisons: $O(n^2)$
  - may be useful for very small input relations
  - may be used with any join-predicate



Page-Oriented Nested-Loops Join

- Input:
  - Relation R
  - Relation S
  - Join-predicate P(r,s)
  - result set RES = {}
- Algorithm:

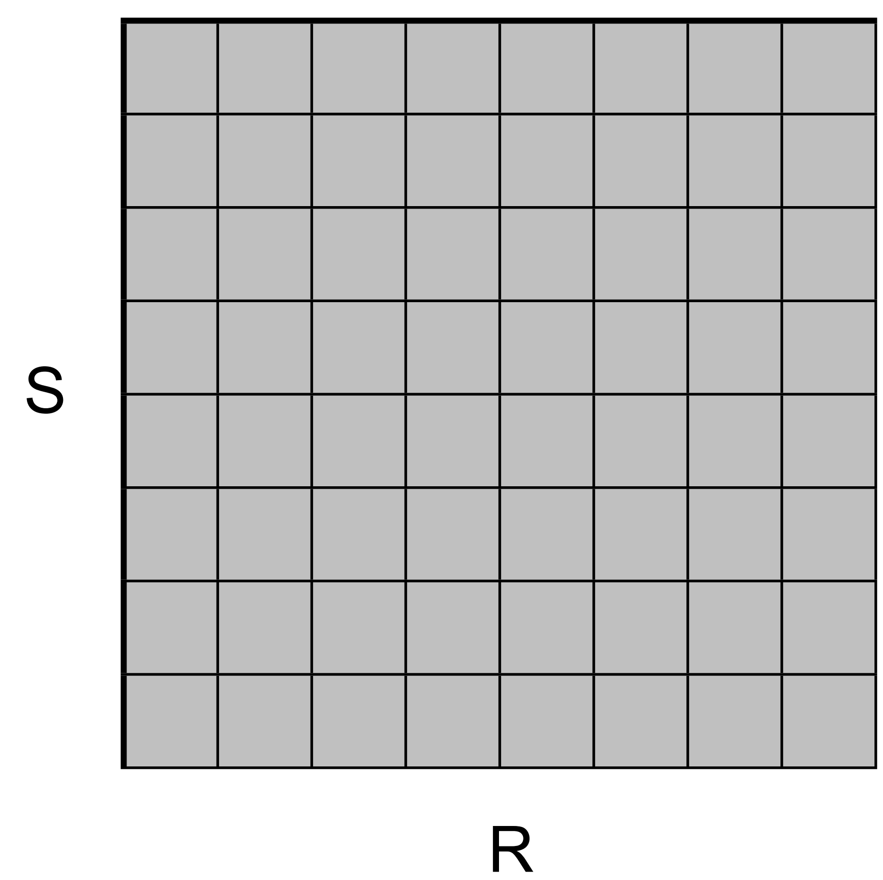
ForEach **page p** of R:

  ForEach s in S:

    ForAll r in p:

      Falls P(r,s):

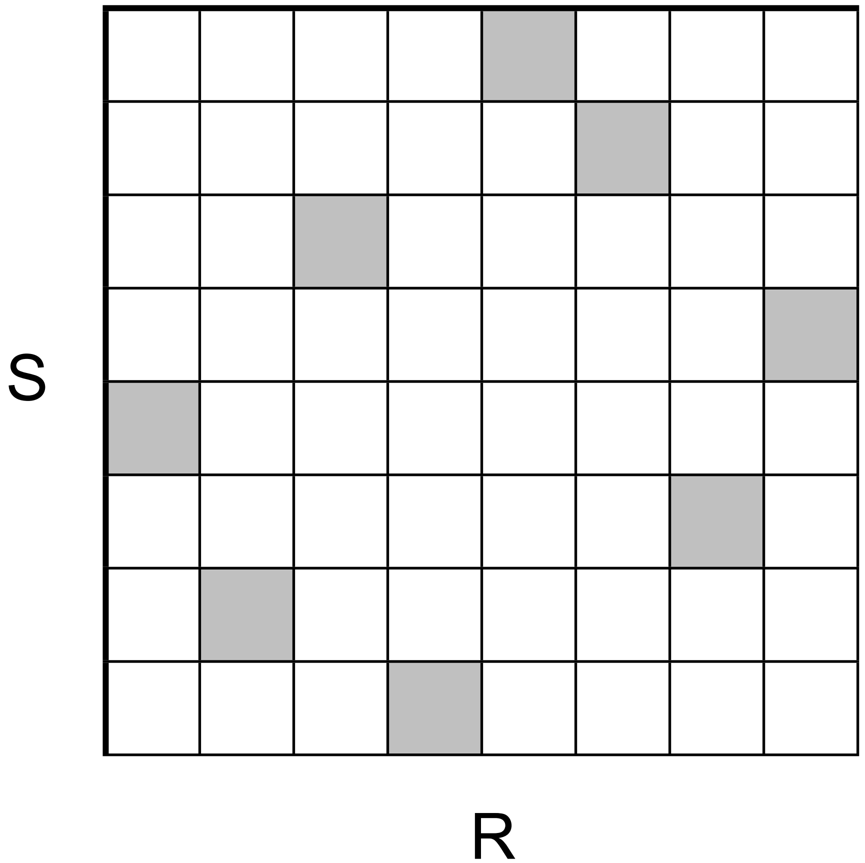
        RES = RES ∪ { (r,s) }



- CPU-effort the same as for simple Nested-Loops Join
- Advantage: much better I/O-behavior
- outer loop will iterate ⌈ |R| / pagesize ⌉ times

Index Nested-Loops Join

- Input:
  - Relation R
  - Relation S
  - result set RES = {}



- Idea: exploit index structure available on one of the input relations
- Algorithm:

  ForEach r in R:

    RES = RES ∪ (r, S.index.query(r))

- Cost: r × cost for index access: O(n log n)
- it may pay-off to create an index before the join in order to be able to perform an index nested-loops join

Sort-Merge Join

- Core idea:
  - sort inputs R and S (using external sorting if necessary) on join attribute
  - scan both sorted input simultaneously and compute records fulfilling the join predicate
- analogy: zip fastener

Example: Sort-Merge Join

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42

Employee (S)			
PersNo.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	32

Example: Sort-Merge Join

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42

ZR

ZS

Employee (S)			
PersNo.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	32

Example: Sort-Merge Join

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42

ZR

ZS

Employee (S)			
PersNo.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	32

Example: Sort-Merge Join

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42

ZR ! ZS

Employee (S)			
PersNo.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	32

Hardware Employee					
ID	Description	PersNo.	Surname	First Name	Age
8	Fire Blade	7	Dittrich	Klaus	43

Example: Sort-Merge Join

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42

ZR

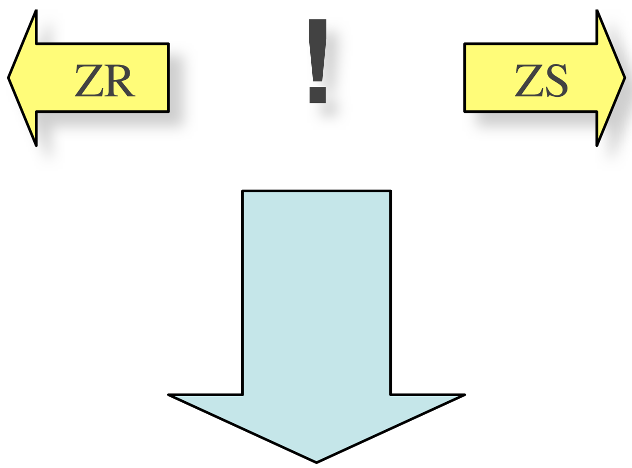
ZS

Employee (S)			
PersNo.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	32

Hardware Employee					
ID	Description	PersNo.	Surname	First Name	Age
8	Fire Blade	7	Dittrich	Klaus	43

Example: Sort-Merge Join

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42



Employee (S)			
PersNo.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	32

Hardware Employee					
ID	Description	PersNo.	Surname	First Name	Age
8	Fire Blade	7	Dittrich	Klaus	43
1	Slow PC	8	Müller	Peter	55

Example: Sort-Merge Join

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42

ZR ! ZS

Employee (S)			
PersNo.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	32

Hardware		Employee			
ID	Description	PersNo.	Surname	First Name	Age
8	Fire Blade	7	Dittrich	Klaus	43
1	Slow PC	8	Müller	Peter	55
12	Fast PC	42	Dittrich	Jens	32

Example: Sort-Merge Join

Hardware (R)

ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42

Employee (S)

PersNo.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	32

ZR ! ZS

Hardware Employee

ID	Description	PersNo.	Surname	First Name	Age
8	Fire Blade	7	Dittrich	Klaus	43
1	Slow PC	8	Müller	Peter	55
12	Fast PC	42	Dittrich	Jens	32
36	Stapler	42	Dittrich	Jens	32

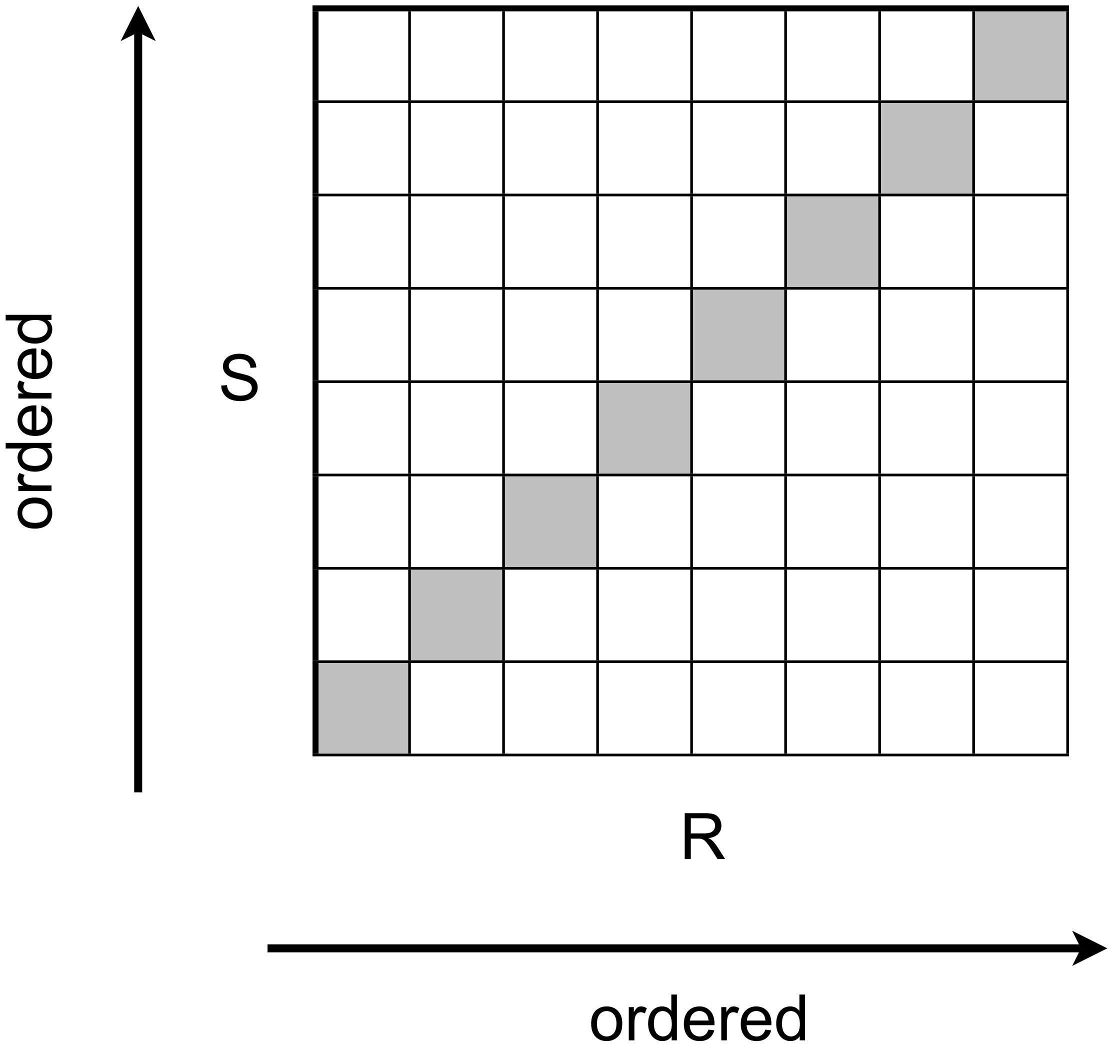
Example: Sort-Merge Join

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42

Employee (S)			
PersNo.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	32

Hardware Employee					
ID	Description	PersNo.	Surname	First Name	Age
8	Fire Blade	7	Dittrich	Klaus	43
1	Slow PC	8	Müller	Peter	55
12	Fast PC	42	Dittrich	Jens	32
36	Stapler	42	Dittrich	Jens	32

Comparison Strategy: Sort-Merge Join



Discussion

- Cost: O(n log n + |RES|)
- sorted inputs may also be delivered by a clustered index
- sorting step may be skipped if a previous operator has already sorted the data (e.g., multiple joins in a query plan)
- combinations possible:
  - one input already implicitly sorted by a clustered index
  - other input explicitly sorted
- this join algorithm may be generalized to work for other join predicates (we will come back to this).
- If none of the join attributes is a primary key, it may happen that pointers have to be moved backwards to ensure that all result tuples are computed.
- This, however, can also be fixed by a more elegant implementation of sort-merge....

Problem: Duplicates in Sort-Merge Join

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42

ZR

ZS

Employee (S)			
No.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	33
42	Hase	Hugo	44

- Correct result (R.ID, S.No)
  - (12, FastPC, 42, Dittrich, Jens, 33)
  - (12, FastPC, 42, Hase, Hugo, 44)
  - (35, Stapler, 42, Dittrich, Jens, 33)
  - (36, Stapler, 42, Hase, Hugo, 44)

Problem: Duplicates in Sort-Merge Join

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42

ZR

ZS

Employee (S)			
No.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	33
42	Hase	Hugo	44

- 1. approach:
  - Nested loops inside groups having the same key
  - Disadvantage
    - at least one of the pointers has to be moved backwards
    - this does not work with the iterator interface!

Problem: Duplicates in Sort-Merge Join

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42

ZR ← → ZS

Employee (S)			
No.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	33
42	Hase	Hugo	44

- 2. approach:
  - “Sweep Area” algorithm
    Literature: Dittrich, Seeger, Taylor, Widmayer, VLDB 2002
  - advantage:
    - iterators never have to be moved backwards
    - approach helps to generalize sort-merge to other non-relational join algorithms
    - correct result in the presence of duplicates is basically for free

Sweep Area Join

- Idea
  - keep two sets S1 and S2 of records that may still produce results
- Algorithm (simplified)

```
Object current = null
S1 = {}, S2 = {}
If (input1.peek() < input2.peek())
    current = input1.next()
    S1 = S1 ∪ {current}                                   insert into sweep area S1
    S2 = S2 \ { s | s ∈ S2, s < current }                 delete from sweep srea S2
    Result = Result ∪ { (current, s) | P(current,s), s ∈ S2 }   query sweep area S2
Else
    current = input2.next()
    S2 = S2 ∪ {current}                                   insert into sweep area S2
    S1 = S1 \ { s | s ∈ S1, s < current }                 delete from sweep area S1
    Result = Result ∪ { (s, current) | P(s, current) s ∈ S1 }   query sweep area S1
```

Sweep Area Join

- Idea
  - keep two sets S1 and S2 of records that may still produce results
- Algorithm (simplified)

```
Object current = null
S1 = {}, S2 = {}
If (input1.peek() < input2.peek())
    current = input1.next()
    S1 = S1 ∪ {current}
    S2 = S2 \ { s | s ∈ S2, s < current }
    Result = Result ∪ { (current, s) | P(current,s), s ∈ S2 }
Else
    current = input2.next()
    S2 = S2 ∪ {current}
    S1 = S1 \ { s | s ∈ S1, s < current }
    Result = Result ∪ { (s, current) | P(s, current) s ∈ S1 }
```

How to implement insert?

How to implement delete?

How to implement query?

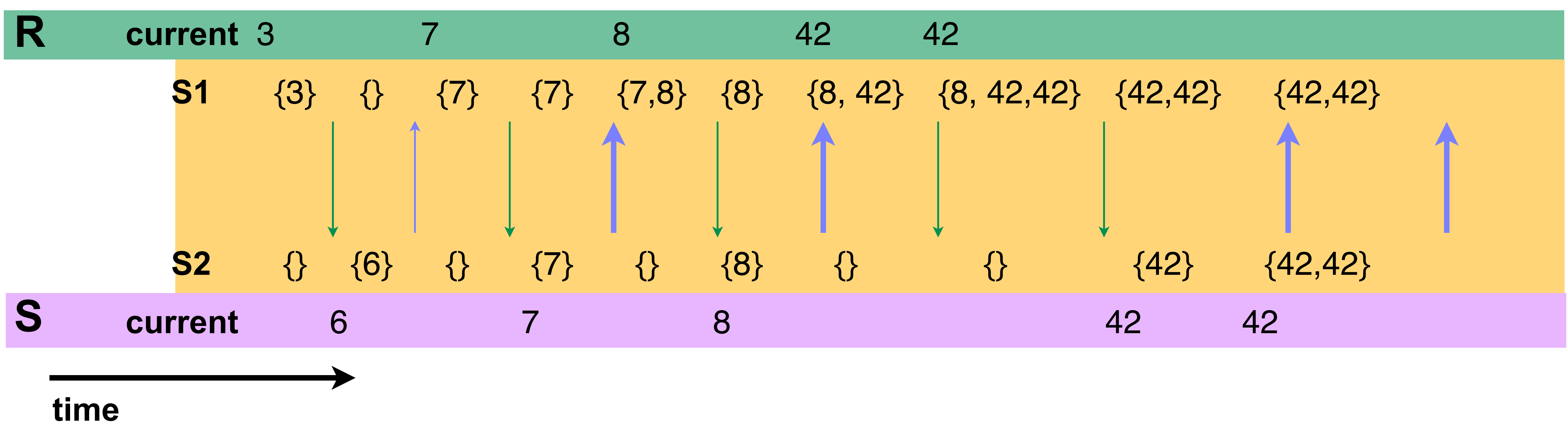
Relational Sweep Area Join (Equi-Join)

- insert
  - use simple list to implement sweep area
  - new elements are appended to the list
- delete
  - scan list from the beginning
  - while listElement < current: delete listElement
- query
  - **All** records contained in the list are part of the result
  - return list.iterator();

Example

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42

Employee (S)			
No.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	33
42	Hase	Hugo	44



Note: in this example only the sweep area is emptied for which the query is executed. Alternatively, one may empty always both sweep areas. In terms of correctness this does not have an impact, however, it may decrease memory usage.

Discussion

- sweep area join may be applied for several other fancy joins
- idea is similar to keeping a **window of interest**
- this window idea is also used in other contexts
- for instance: stream processing
  - infinite stream of elements
  - compute average over items seen in the past 2 hours
  - this is realized by keeping a **data window**
- we will return to sweep areas in the context of non-relational join processing

Simple Hash-Join

- Assumption:
  - R is smaller than S, i.e., |R| < |S|
  - R fits into main memory
- Algorithm:

  read R into main memory
  create a hash table on R
  read table S sequentially
  ForEach s in S:
      RES = RES ∪ { s, R.hash_table.probe(s) }

What happens if R is larger than the available main memory?

Grace Hash-Join

- Algorithm:

  partition smaller table T ∈ { R,S }, such that each partition T_i of T fits into main memory

  T is called **build input** or **inner table**

  partition Q ∈ { R,S }, Q ≠ T using the same partitioning function into partitions Q_i (a pair $(T_i,Q_j),i=j$ denotes corresponding partitions)

  // this means that Q and T are partitioned in a way such that holds:

  // when joining T_i and Q_j $i \neq j$ no join result may be found

  Q is called **probe input** or **outer table**

  ForEach partition pair (T_i,Q_j):

  Perform a simple hash-join

Partitioning Details

- Inputs:
  - input R
- Required:
  - function computing for any given tuple r in R its partition
- Important:
  - partitioning has to be done based on the join key! (otherwise the invariant $(R_i, S_j) = \{\}$ for $i \neq j$ will be violated)
  - round robin or similar on TID does **not** work!

Interval Partitioning

M=3

- Idea: partition join key into intervals

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42

			PersNo
R_1 =	4 | Cray | 3		1-6
R_2 =	8 | Fire Blade | 7 1 | Slow PC | 8		7-12
R_3 =	{}		13-18
R_4 =	{}		19-24
R_5 =	{}		25-30
R_6 =	{}		31-36
R_7 =	12 | Fast PC | 42 36 | Stapler | 42		37-42

Hash-Partitioning

M=3

- Idea: use a hash-function on the join key modulo number of partitions
- Goal: hash-function should provide a balanced distribution of data to partitions
- example for a good hash function: see previous slides on hashing
- let W be the number of partitions
- rule of thumb: $W := \lceil R/(M-1) \times F \rceil$, $F \approx 1.2$

  (if this does not work to obtain a partitioning: recursive partitioning)

Hash-Partitioning

M=3
W=3

- Idea: use a hash-function on the join key modulo number of partitions

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42

Partition	ID	Description	PersNo.	rand(PersNo)	%3
R_1 =	12	Fast PC	42	19488	0
	36	Stapler	42	19488	0
R_2 =	1	Slow PC	8	14218	1
	4	Cray	3	17893	1
R_3 =	8	Fire Blade	7	15467	2

Complete Join: Phase 1

M=3
W=3

- partition **R**

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42

	ID	Description	PersNo.	rand(PersNo)	%3
R_1 =	12	Fast PC	42	19488	0
	36	Stapler	42	19488	0
R_2 =	1	Slow PC	8	14218	1
	4	Cray	3	17893	1
R_3 =	8	Fire Blade	7	15467	2

Complete Join: Phase 1

M=3
W=3

- partition **S**

Employee (S)			
PersNo.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	32

PersNo

	PersNo	Surname	First Name	Age	**rand(PersNo)**	**%3**
S_1 =	42	Dittrich	Jens	32	19488	0
S_2 =	8	Müller	Peter	55	14218	1
S_3 =	7	Dittrich	Klaus	43	15467	2
	6	Meier	Hans	37	18488	2

Complete Join: Phase 2, Pair (R_1,S_1)

1. load first partition R_1 of R into main memory and create hash table:

R_1 =

12	Fast PC	42
36	Stapler	42

HS

2. probe each tuple of S_1 against hash table

S_1 =

42	Dittrich	Jens	32

probe of S_1 against hash table computes the result:

12	Fast PC	42	Dittrich	Jens	32
36	Stapler	42	Dittrich	Jens	32

Complete Join: Phase 2, Pair (R_2,S_2)

1. load second partition R_2 of R into main memory and create hash table:

R_2 =

1	Slow PC	8
4	Cray	3

HS

2. probe each tuple of S_2 against hash table

S_2 =

8	Müller	Peter	55

probe of S_s against hash table computes the result:

1	Slow PC	8	Müller	Peter	55

Complete Join: Phase 2, Pair (R_3,S_3)

1. load third partition R_3 of R into main memory and create hash table: R_3 =

8	Fire Blade	7

HS

2. probe each tuple of S_3 against hash table

S_3 =

7	Dittrich	Klaus	43
6	Meier	Hans	37

probe of S_s against hash table computes the result:

8	Fire Blade	7	Dittrich	Klaus	43

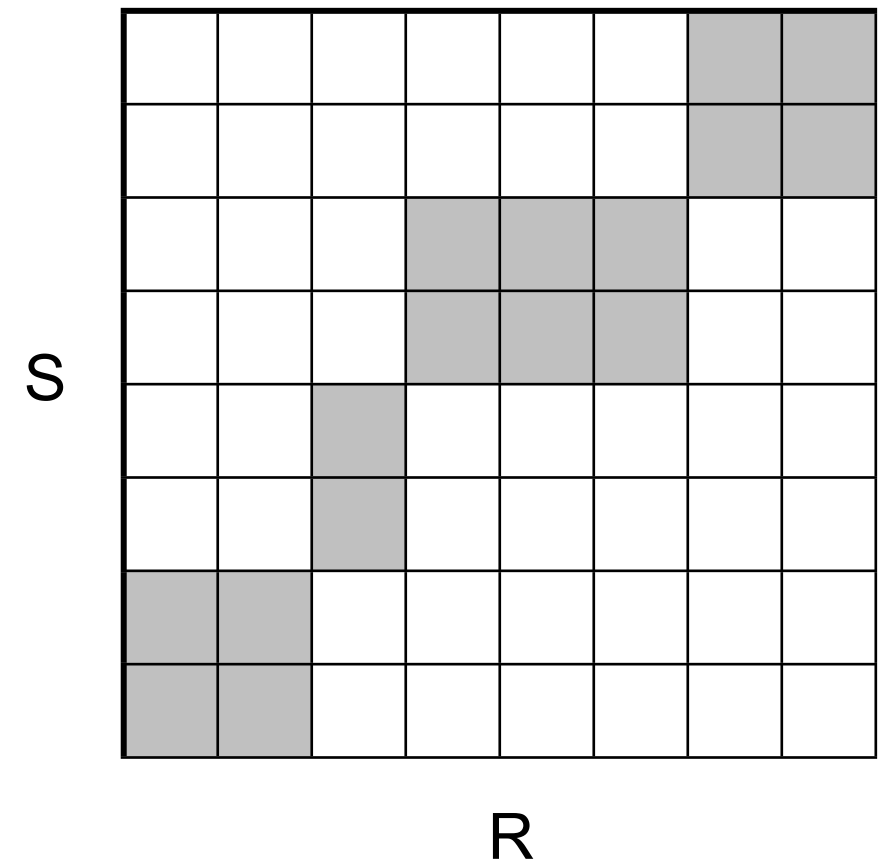
Result of the Join

(R_1,S_1)	12	Fast PC	42	Dittrich	Jens	32
	36	Stapler	42	Dittrich	Jens	32
(R_2,S_2)	1	Slow PC	8	Müller	Peter	55
(R_3,S_3)	8	Fire Blade	7	Dittrich	Klaus	43

- Note: records are not reported in join key order
- as we did at no point sort the data!

Discussion

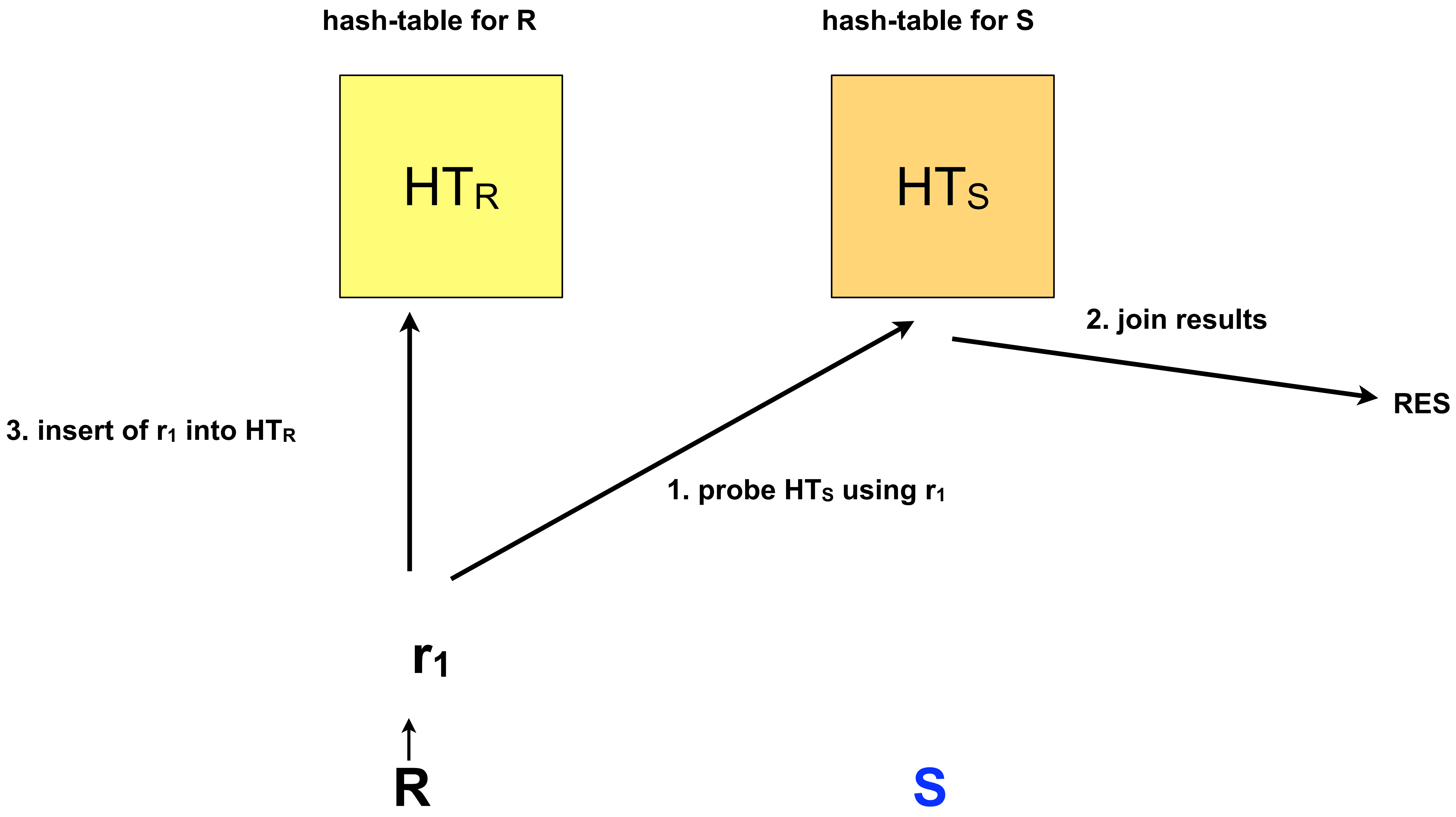
- always use smaller relation as build input
- Grace Hash Join wins in many cases against all other join algorithms
- but: grace hash join only works for equi-joins!
- some enhancements:
  - use a bloom-filter to prefilter outer tables
  - if possible: merge small partitions into bigger ones => avoids having zillions of partitions



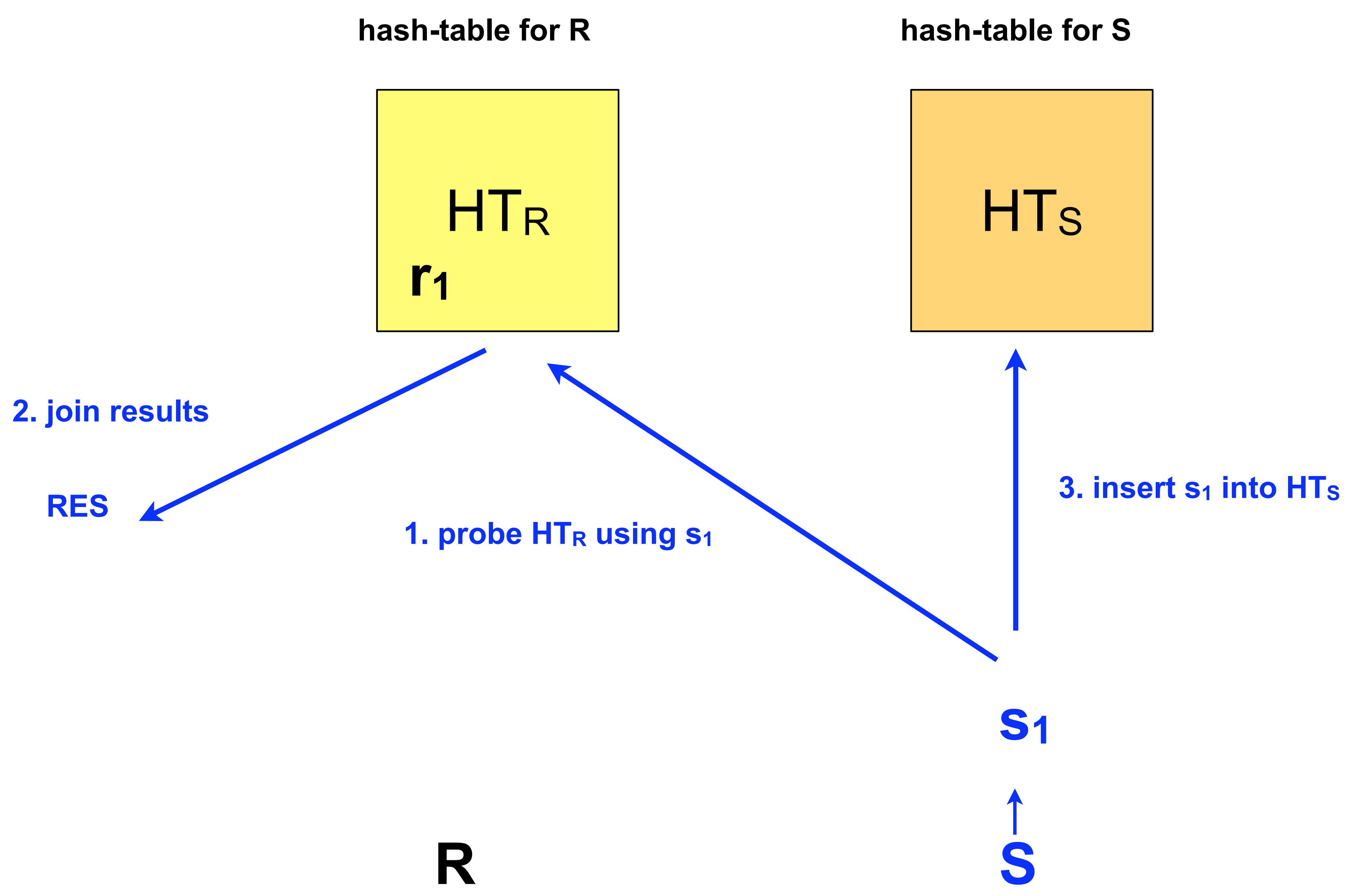
Double-pipelined Hash Join

- Idea: keep two hash-tables in main memory, HT_R for elements of R, HT_S for elements of S
- draw alternately a tuple from R and S
- for example, if we draw an element r of input R
  - perform a probe(r) on hash-table HT_S, in order to find all join mates
  - insert r into HT_R
- analogue for tuples s from input S
- Discussion
  - delivers results early => no blocking
  - works well for distributed systems and whenever pipelining is needed
  - works well if no statistics available for input data
  - extensions necessary when main memory overflows (XJoin)

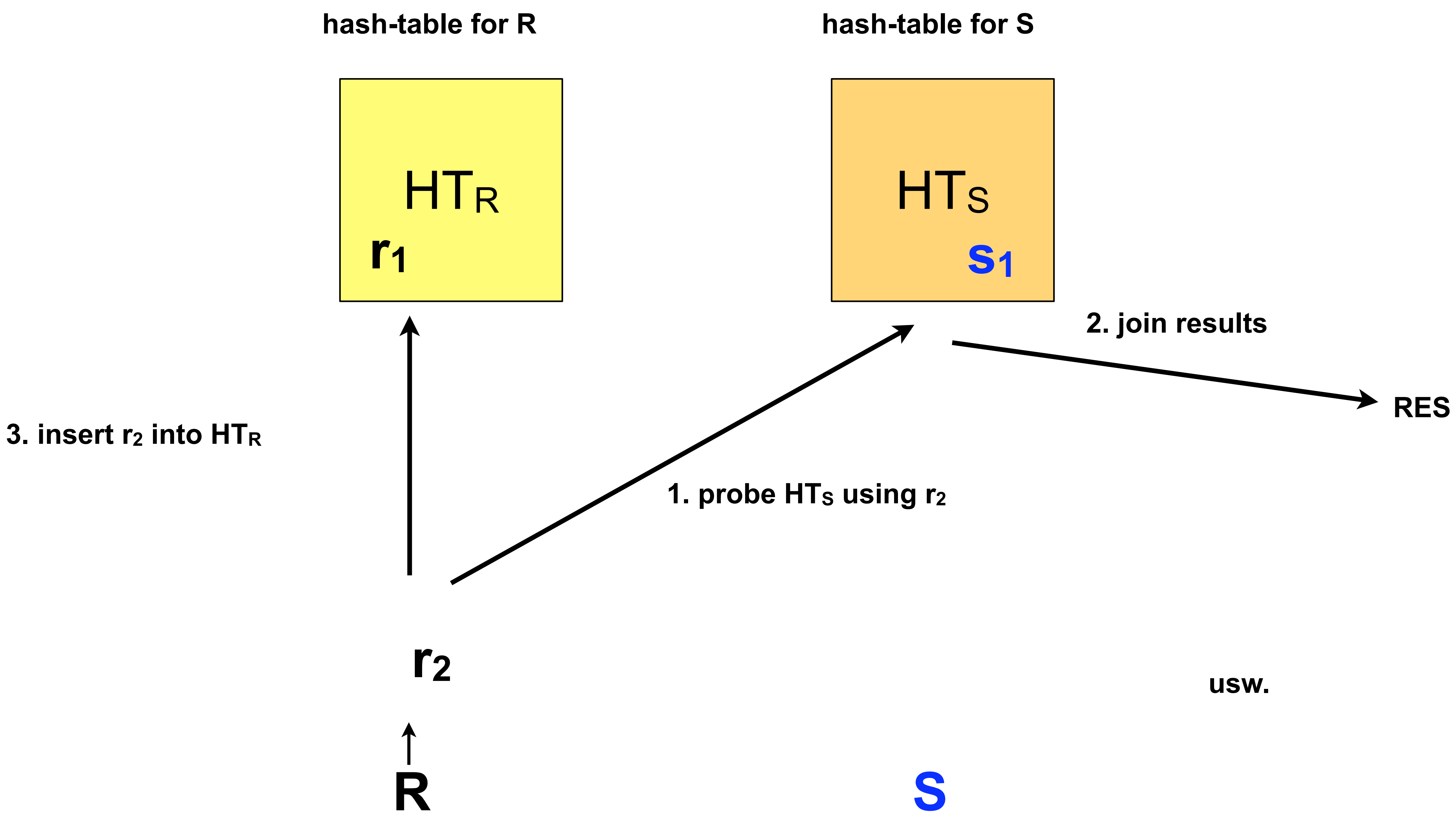
Example: Double-pipelined Hash Join



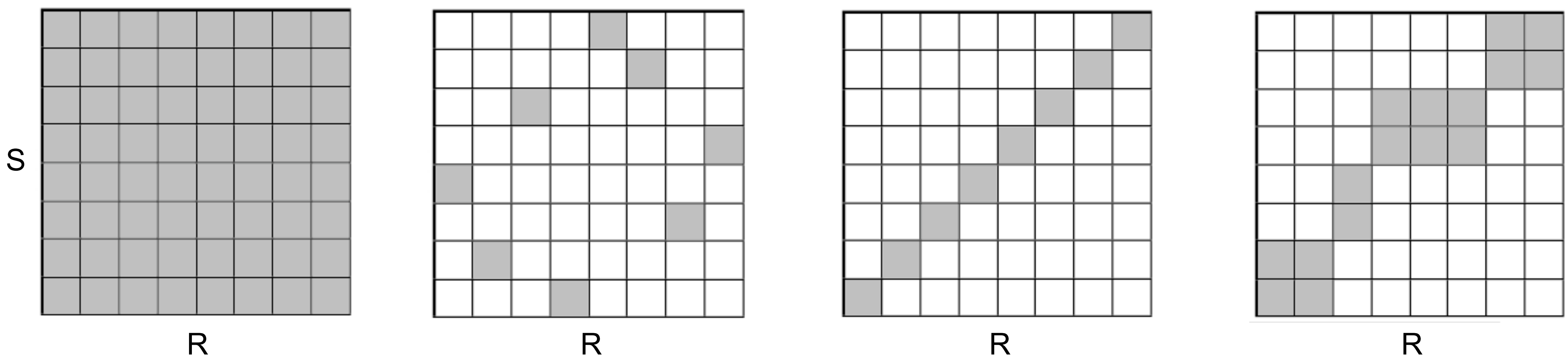
Example: Double-pipelined Hash Join



Example: Double-pipelined Hash Join



Overview: Comparison Strategies



Nested Loops | Index Nested Loops | Sort Merge | Hash

Literature: Priti Mishra, Margaret H. Eich: Join Processing in Relational Databases. ACM Computing Surveys 24(1), 1992: 63-113

Discussion

- Many of these join techniques can be extended to other types of data, e.g., spatial and temporal data, XML
- hashing => use techniques discussed in hash and partition based indexing
- sorting => use techniques discussed for linearization
- join techniques may be extended to outer joins (left, right, full)
- join techniques can be used to implement operators like intersection

How?

Operator Implementations.

Aggregation and Grouping.

Aggregation and Grouping

- Examples
  - AVG, MIN, MAX (blocking)
  - Distinct (non-blocking)

- 4 classes of algorithms
  - nested-loops
  - sort-based
  - hash-based
  - early aggregation

Nested Loops: Distinct

- Inputs: input R, output R’, set T = {}
- **Algorithm**

```
ForEach r in R:
    boolean contained = false;
    ForEach t in T:
        If t == r:
            contained = true;
            break;
    If !contained:
        write r to R’
        T = T ∪ {r}
```

- Discussion
  - Advantage: non-blocking, results are passed on immediately
  - Disadvantage: quadratic cost

Hash-Based Distinct

- like nested-loops, but: replace inner loop by a hash table ht
- **Algorithm**

```
ForEach r in R:
    If !ht.contains(r):
        write r to R'
        ht.insert(r)
```

- Discussion
  - Advantages
    - non-blocking
    - results are passed on immediately
    - efficient: O(n)
  - Disadvantage
    - what happens if hash table ht does not fit into main memory? (algorithm similar to Grace Hash Join)

Sort-Based Distinct

- **Algorithm**

```
T‘ = sort(R);
lastElement = T’.top();
write lastElement to R’;
ForEach r in T’:
    If r != lastElement:
        lastElement = r;
        write r to R’;
```

- Discussion
  - Advantages
    - easy to implement and efficient: O(n log n)
    - also works for cases where R‘ does not fit into main memory (in contrast to hash-based distinct)
  - Disadvantage: sorting breaks/blocks pipelining

Early Aggregation

- core idea: records belonging to the same group may already be aggregated during sorting or partitioning
- For sorting:
  - aggregate during replacement selection (while outputting elements)
  - aggregate during merges
- For partitioning:
  - keep hash table in main memory to preaggregate (as before)
  - if records are read that do not belong to groups in the hash table and hash table may not be increased:
    write these records to an overflow file
  - when all elements from input have been read:
    - output groups from hash table
    - repeat algorithm recursively using overflow file as an input
- Advantage for both sorting and partitioning: less data, less I/O

Operator Implementations.

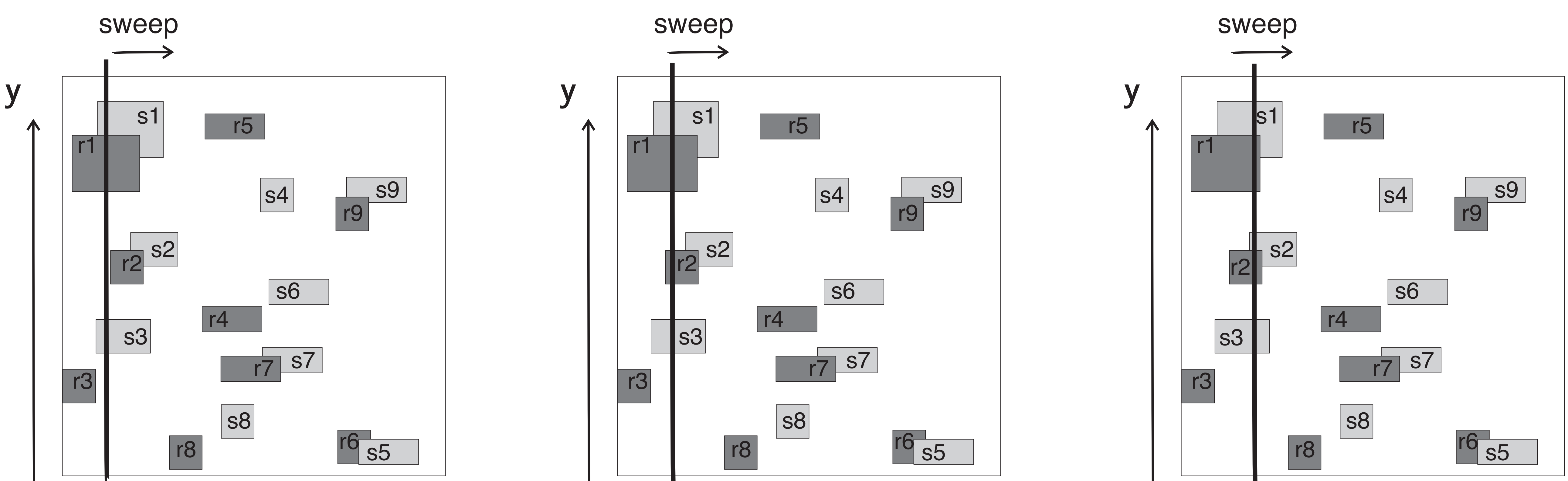
Non-Relational Join Algorithms.

Core Ideas

- use either hashing/partitioning
  - build index, i.e., hash table, partitioning, or grid at query time
  - throw index away after join completion
- or sorting plus merge
  - sort inputs at query time
  - then merge
- use ideas from multi-dimensional index structures to implement this

Spatial Joins: Plane Sweep

- Input: 2 sets of rectangles R and S
- Result: all pairs of rectangles that intersect



state R:	state S:	state R:	state S:	state R:	state S:
r1	s1, s3	r1, r2	s1, s3	r1, r2	s1, s3, s2

Plane Sweep using a Sweep Area

- insert
  - implement sweep area as a one-dimensional interval-tree (see Cormen et.al.)
  - new elements will be inserted w.r.t. their y-interval
- delete
  - use a separate heap
  - priority: max value of x-interval (ascending)
  - remove all elements from the heap where top-Element < current (max value of x-interval smaller current x-position)
- query
  - **all** intervals contained in the sweep area
    and intervals whose y-interval intersects y-interval of the query's y-interval

z-Code Join

- **Algorithm**

  partition data using a space-filling curve

  ForEach r in R:

      For each partition p intersected by r:

          write entry e= (zcode(p), r) to R’

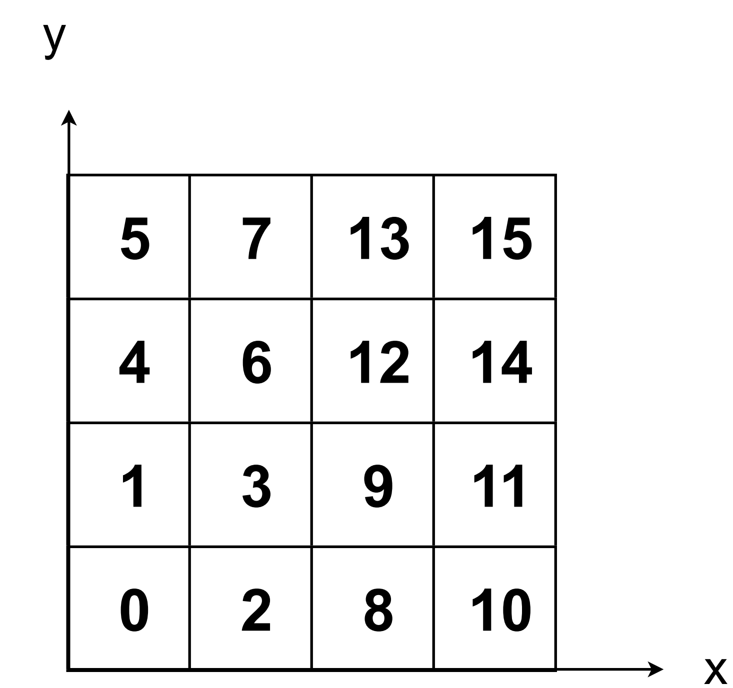
  R’’ = sort(R’)   //sort key = z-code

  do the same for input S

  merge R’’ and S’’ considering all pairs of entries

      $e_R \in$ R’’, $e_S \in$ S’’ where

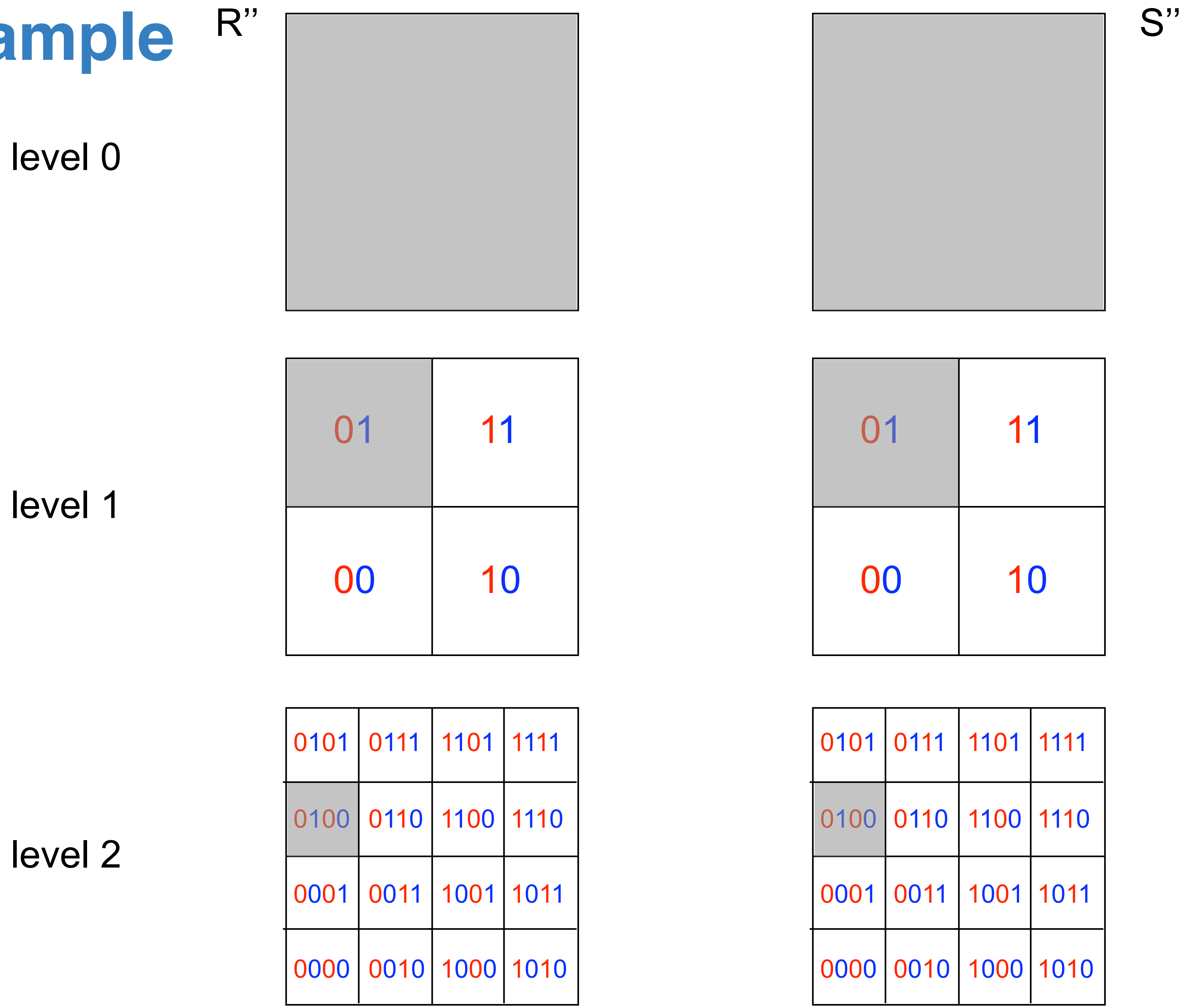
          e_R.zcode and e_S.zcode have a common prefix



z-order

Note: algorithmically this is still the same idea as for the equi and spatial join. We just extended our understanding of a “sweep area“.

Example


R''
S''
level 0
level 1
01
11
00
10
01
11
00
10
level 2
0101
0111
1101
1111
0100
0110
1100
1110
0001
0011
1001
1011
0000
0010
1000
1010
0101
0111
1101
1111
0100
0110
1100
1110
0001
0011
1001
1011
0000
0010
1000
1010

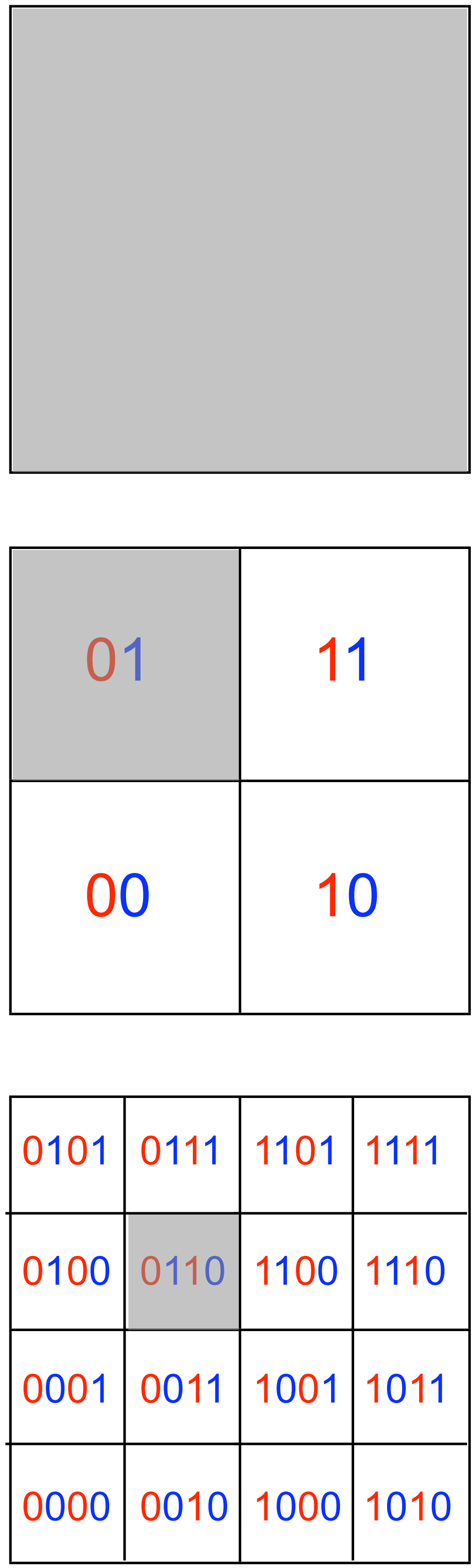
Example

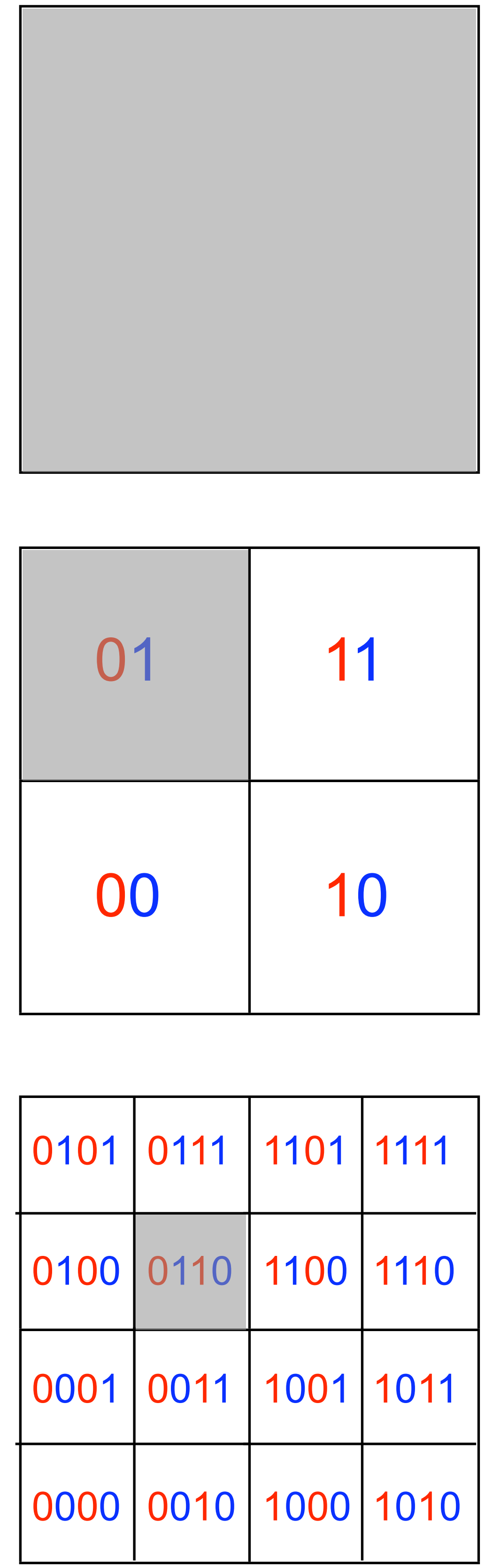
level 0

level 1

level 2

R''





S''

next <current> element in S'':
0110

Effect
0110 removes all elements on this level having a smaller z-value

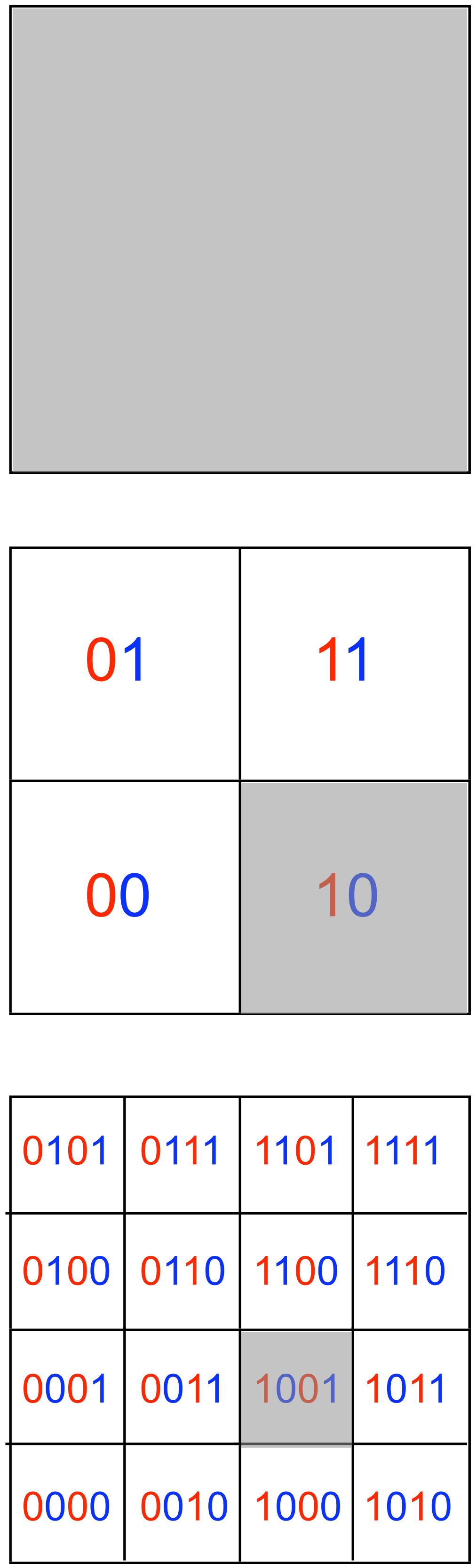
Example

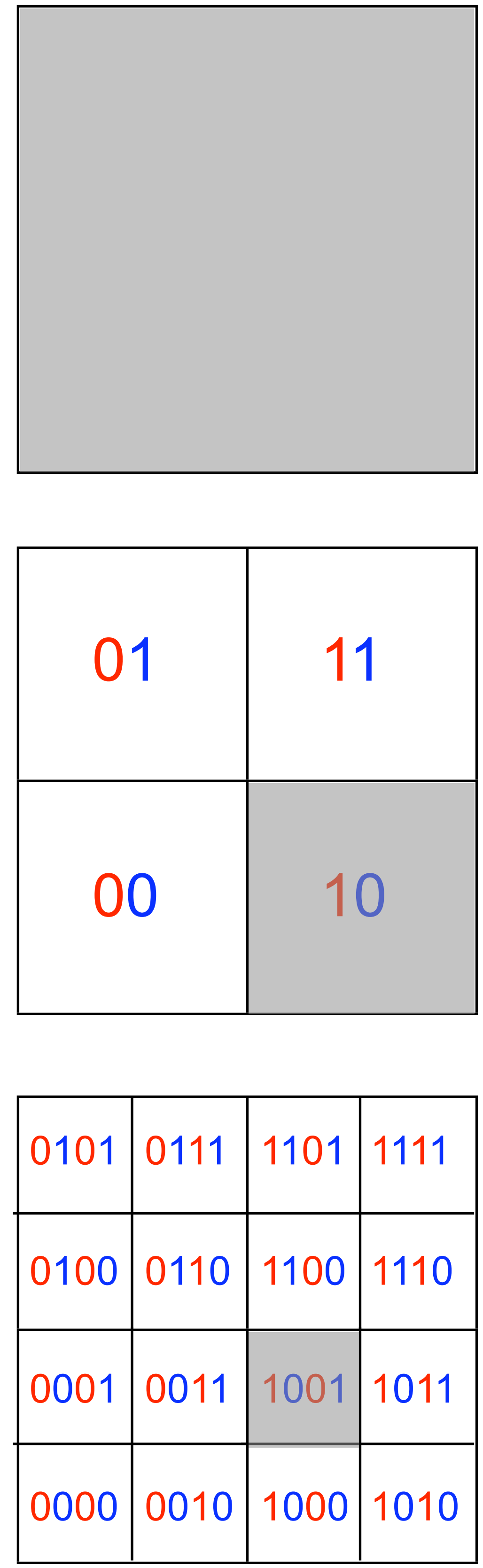
level 0

level 1

level 2

R’’





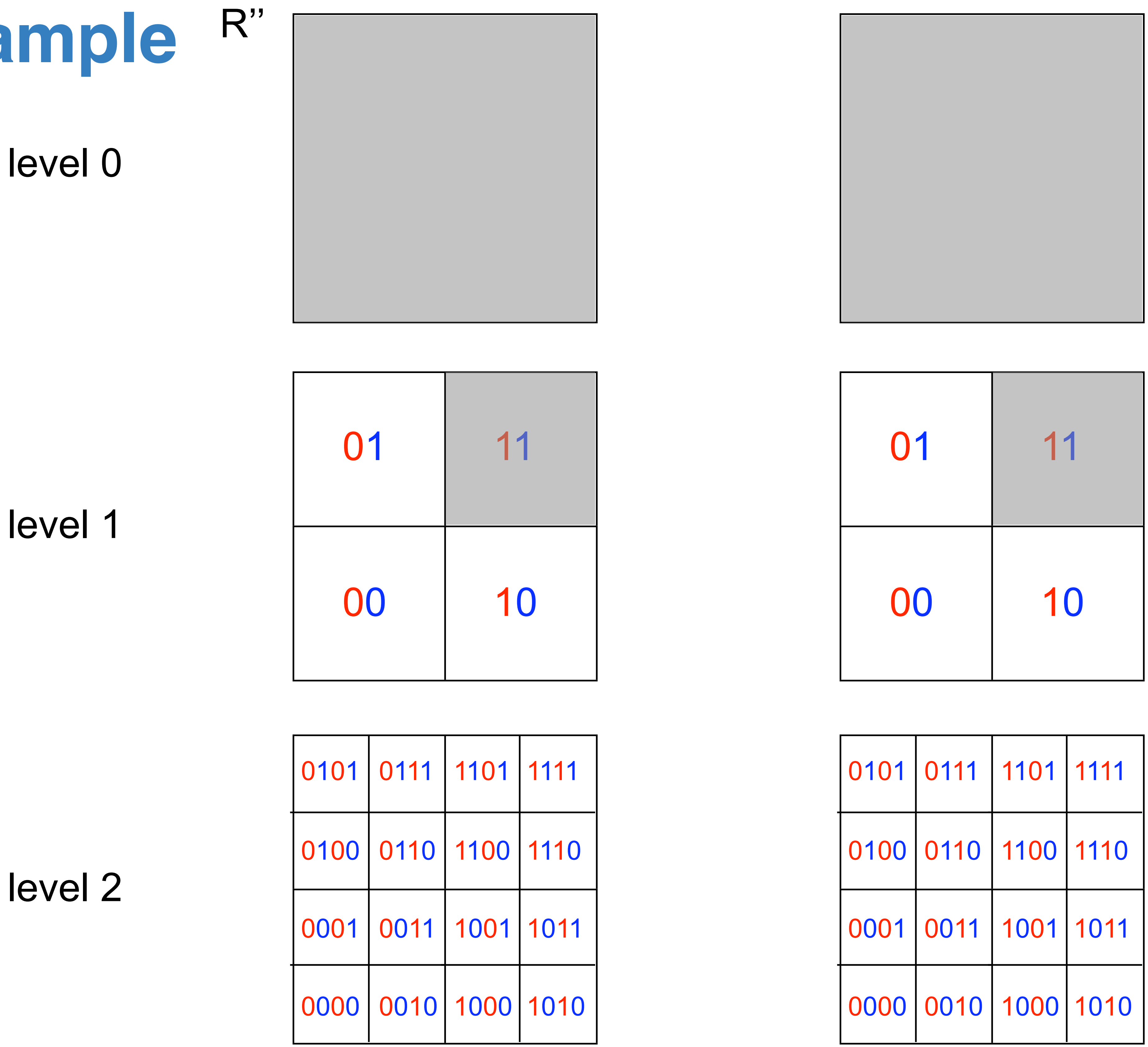
S’’

next <current> element in S’’:
1001

Effect
0110 removes all elements on this level having a smaller z-value

In addition:
all elements having a smaller prefix on lower levels will be also removed.

Example



S''

next <current> element in S'':
11

Effect
11 removes all elements having a z-code of length 2 or longer

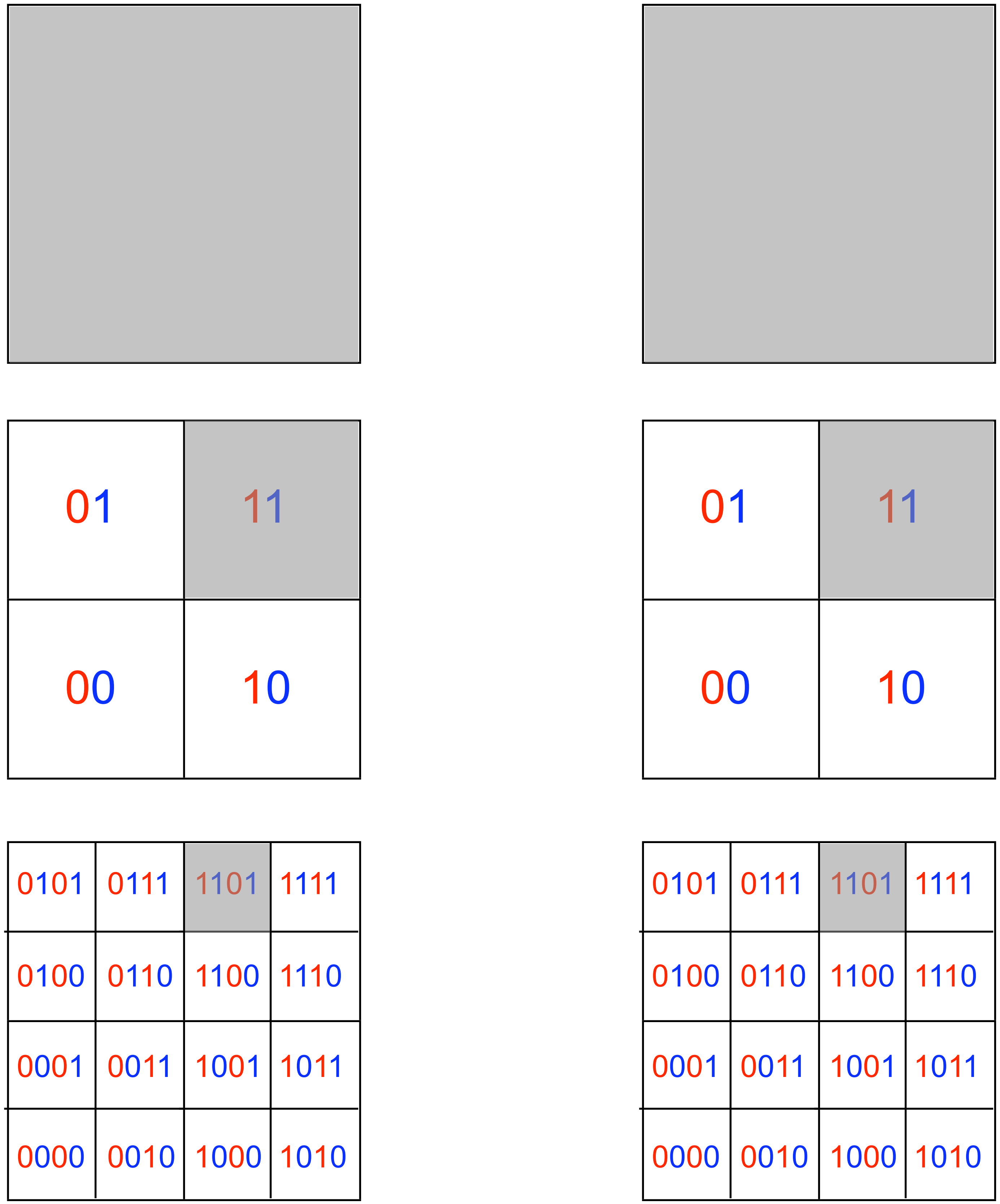
Example

R''

level 0

level 1

level 2



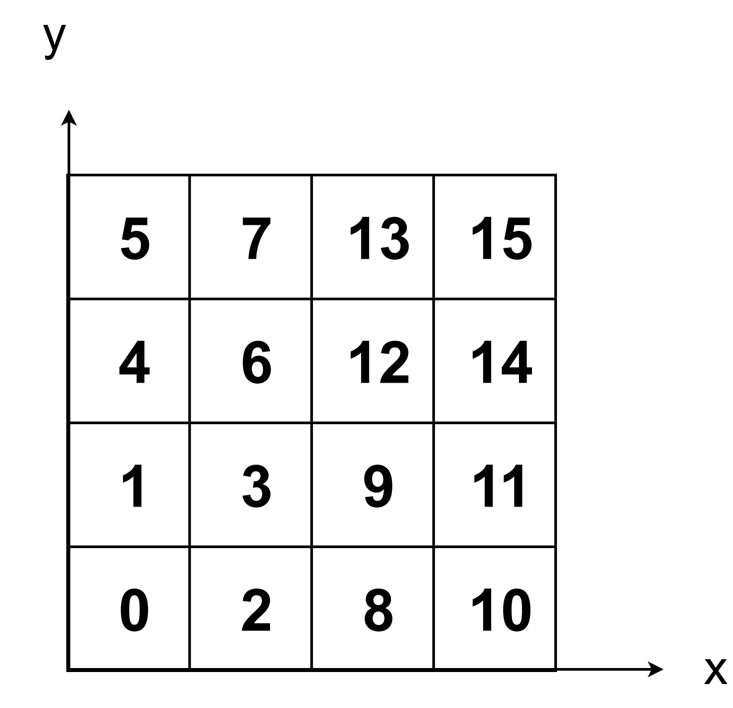
S''

next <current> element in S'':
1101

Effect
1101 does not remove any elements

z-Code Join: Discussion

- also works for more than 2 dimensions
- algorithm keeps two paths with the currently treated z-codes in main memory
- 2 cases:
  - z-codes have different length (partitions have different size)
  - z-codes have the same length (partitions have the same size)



z-order

Question: How to implement this?

Discussion

- For each level we need to keep at most one set of elements corresponding to one z-code.
- join simulates a **synchronous tree traversal**
- therefore this is more like a join of two tree structured indexes
- drawbacks:
  - still square complexity
  - big rectangles may kill performance
  - requires extra tricks that may be hard to tune

z-Code Join using a Sweep Area

- **insert**
  - implement sweep area as a list of sets
  - one set S_i for each length L of a z-code (S_L,..., S_0)
  - new elements are inserted w.r.t. the length of their z-code
  - **Note**: each set will at any time be represented by a **single** z-code! (Somewhat similar to the condition in an equi join: there may be only one key in the sweep area if we always delete both sweep areas.)
- **delete**

  let L be the length of the z-code of current

  empty all sets where length of z-code > L

  ForEach set s in S_L to S_0:

  If z-code(s) and z-code(current) do not have a common prefix w.r.t. length L:

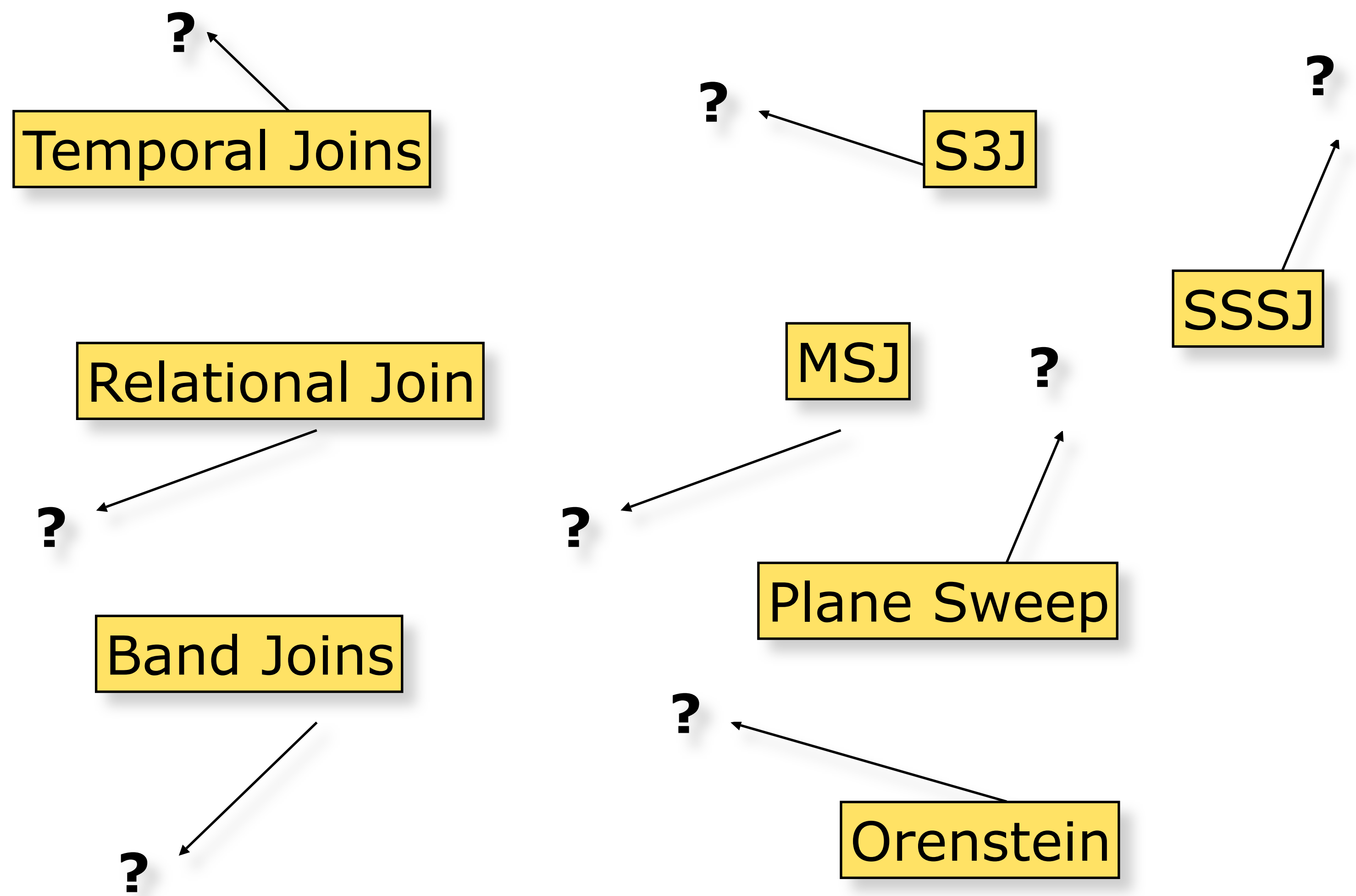
  empty s

  Else break
- **query**
  - **All** entries contained in the sweep area (similar to equi join)

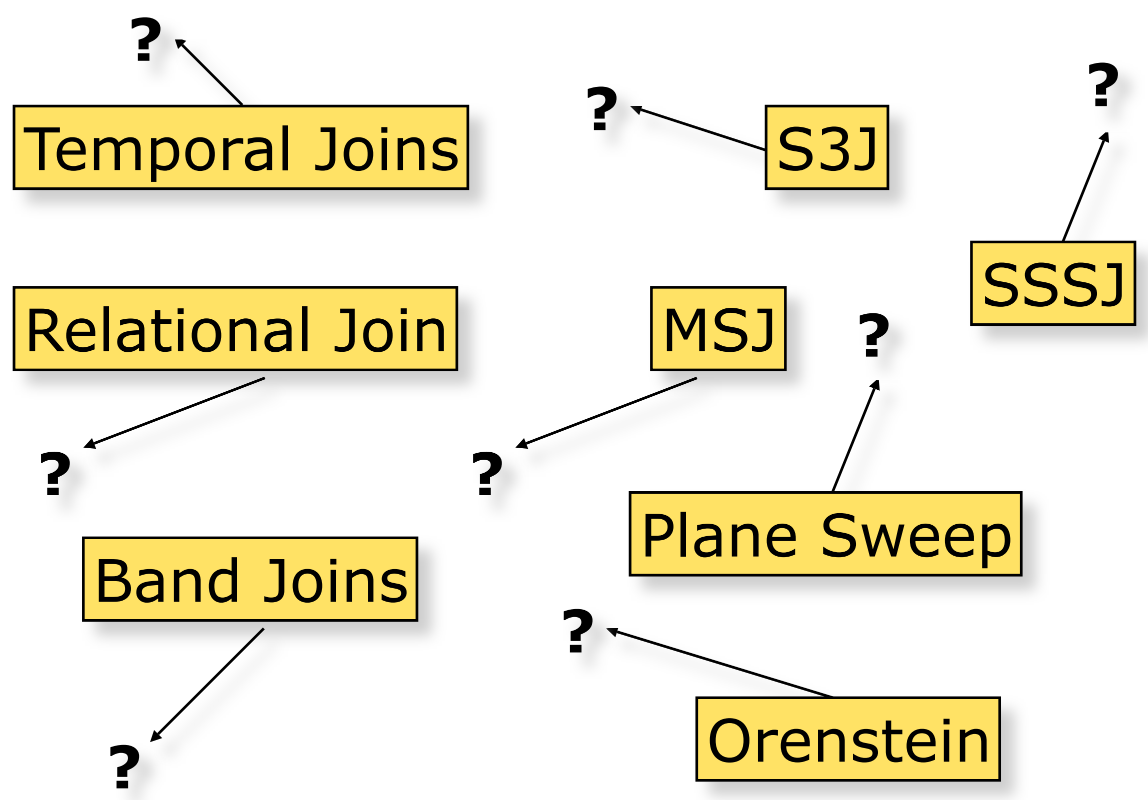
Note: alternative implementation using a stack possible

Problem

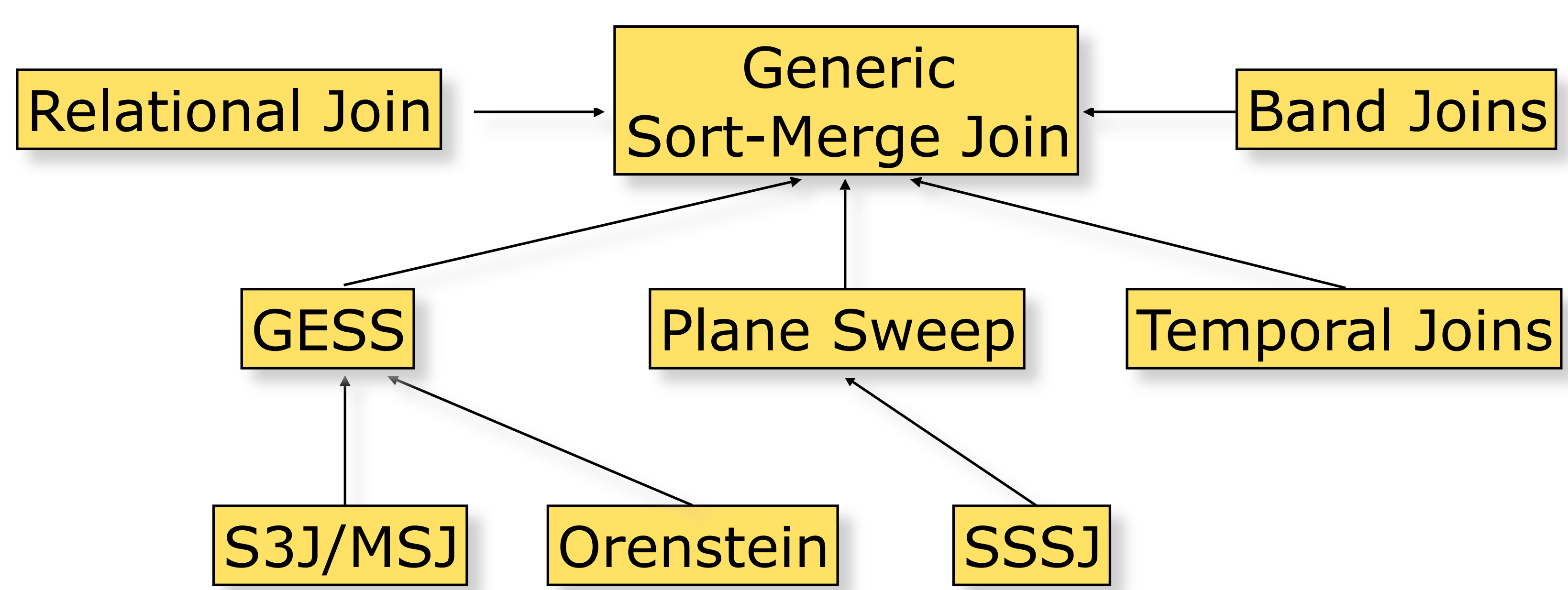
- many non-standard join algorithms are based on a similar idea
- implementations and descriptions, however, are monolithic



- Drawbacks: code duplication, hard to maintain/debug, optimizations have to be coded for each join algorithm separately, etc...



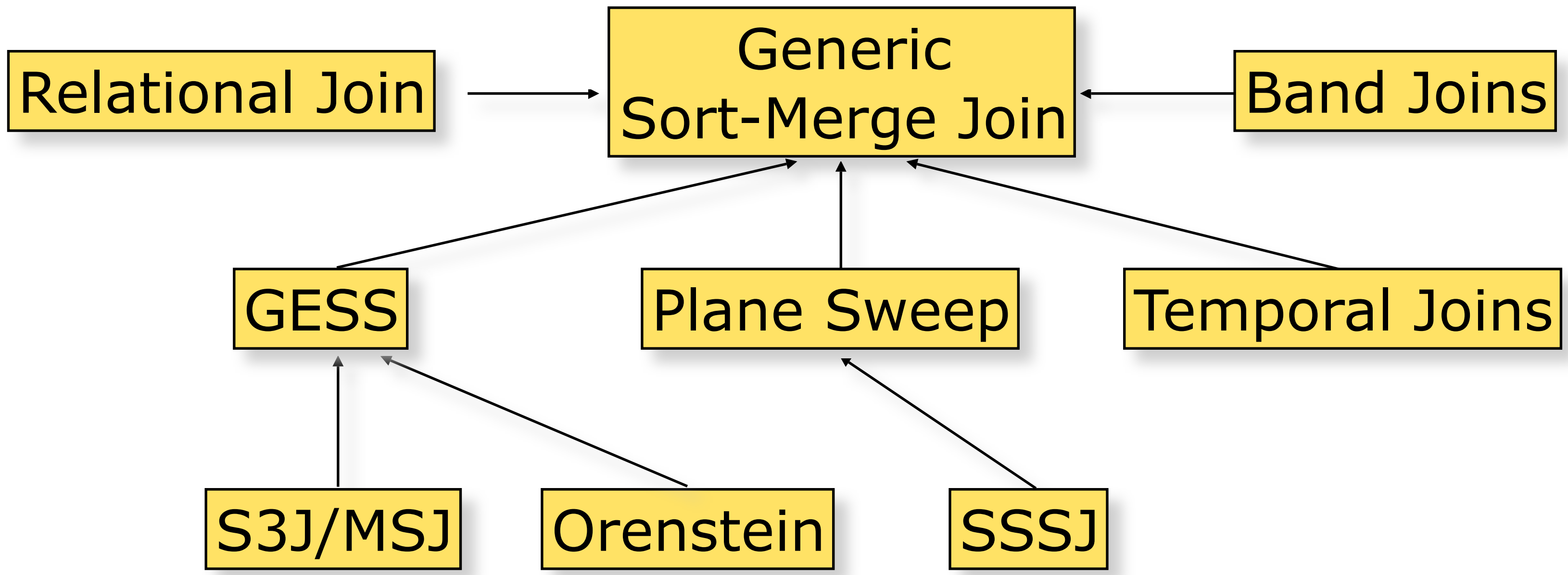
Unified Join-Framework



Solution:

one generic framework for all sort-based join algorithms

Advantages



- no code duplication similar to GIST/XXL approach
- extensions only implement delta to core idea of generic sort-merge join
- improvements made for the generic sort-merge join are automatically available for all subclasses
- core technique for unification: **sweep areas**!
- Literature:
  - Jens-Peter Dittrich, Bernhard Seeger: GESS: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces. KDD 2001
  - Jens-Peter Dittrich, Bernhard Seeger, David Scot Taylor, Peter Widmayer: Progressive Merge Join: A Generic and Non-Blocking Sort-Based Join Algorithm. VLDB 2002

Next Topic: Query Optimization.