COMP 421: Files & Databases

Lecture 13: Query Execution

UNC DEPARTMENT OF COMPUTER SCIENCE

Announcements

Midterm 1: Done grading, just a couple of make-ups still in the pipeline.

Monday: we'll discuss/release exam grades, mid-semester grades

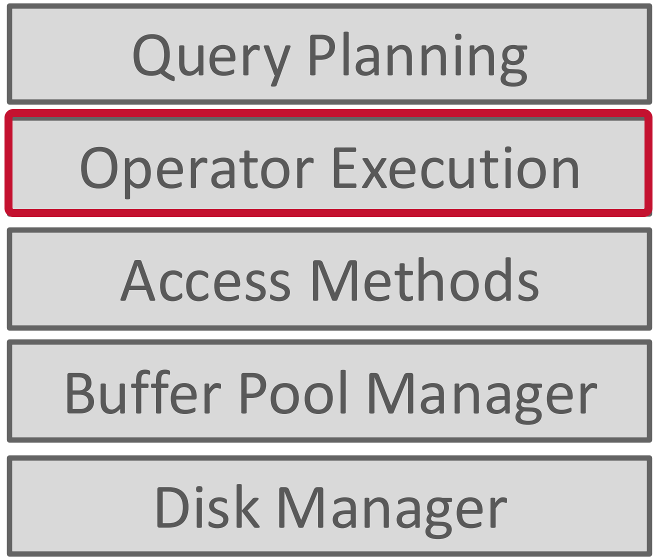
Exam Solutions: we'll either release next week or make available in office hours

Last Class

Operator Execution

- How to translate relational operators into fast code
- Good news: one of the hard ones, sorting, we've already done
- **Today**: the other hard one, **joins**

Goal: want fast, I/O efficient operators to use when we get to query planning and execution



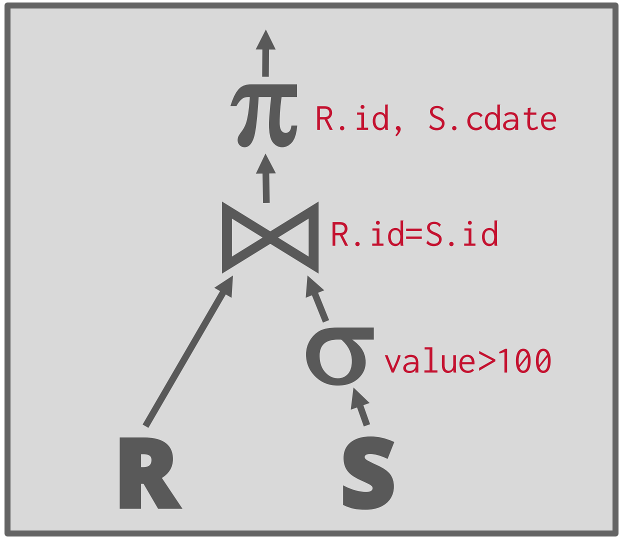
Query Plan

The operators are arranged in a tree.

Data flows from the leaves of the tree up towards the root.
→ We will discuss the granularity of the data movement next lecture.

The output of the root node is the result of the query.

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Today's Agenda

Processing Models

Access Methods

Modification Queries

Expression Evaluation

Processing Model

A DBMS's **processing model** defines how the system executes a query plan and moves data from one operator to the next.

→ Different trade-offs for workloads (OLTP vs. OLAP).

Each processing model is comprised of two types of execution paths:

→ **Control Flow:** How the DBMS invokes an operator.

→ **Data Flow:** How an operator sends its results.

The output of an operator can be either whole tuples (NSM) or subsets of columns (DSM).

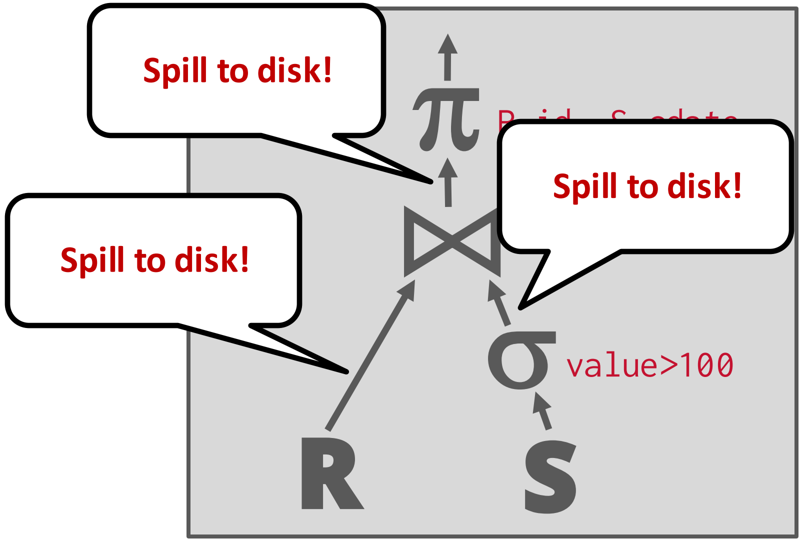
Query Execution

A query plan is a DAG of **operators**.

Naïve idea: look for an operator whose children are complete, run this operator until done

Problem: what to do with intermediate results?

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Processing Model

Approach #1: Iterator Model ⬅ ***Most Common***

Approach #2: Materialization Model ⬅ ***Rare***

Approach #3: Vectorized / Batch Model ⬅ ***Common***

Iterator Model

Each query plan **operator** implements a `Next()` function.

→ On each invocation, the operator returns either a single tuple or a EOF marker if there are no more tuples.

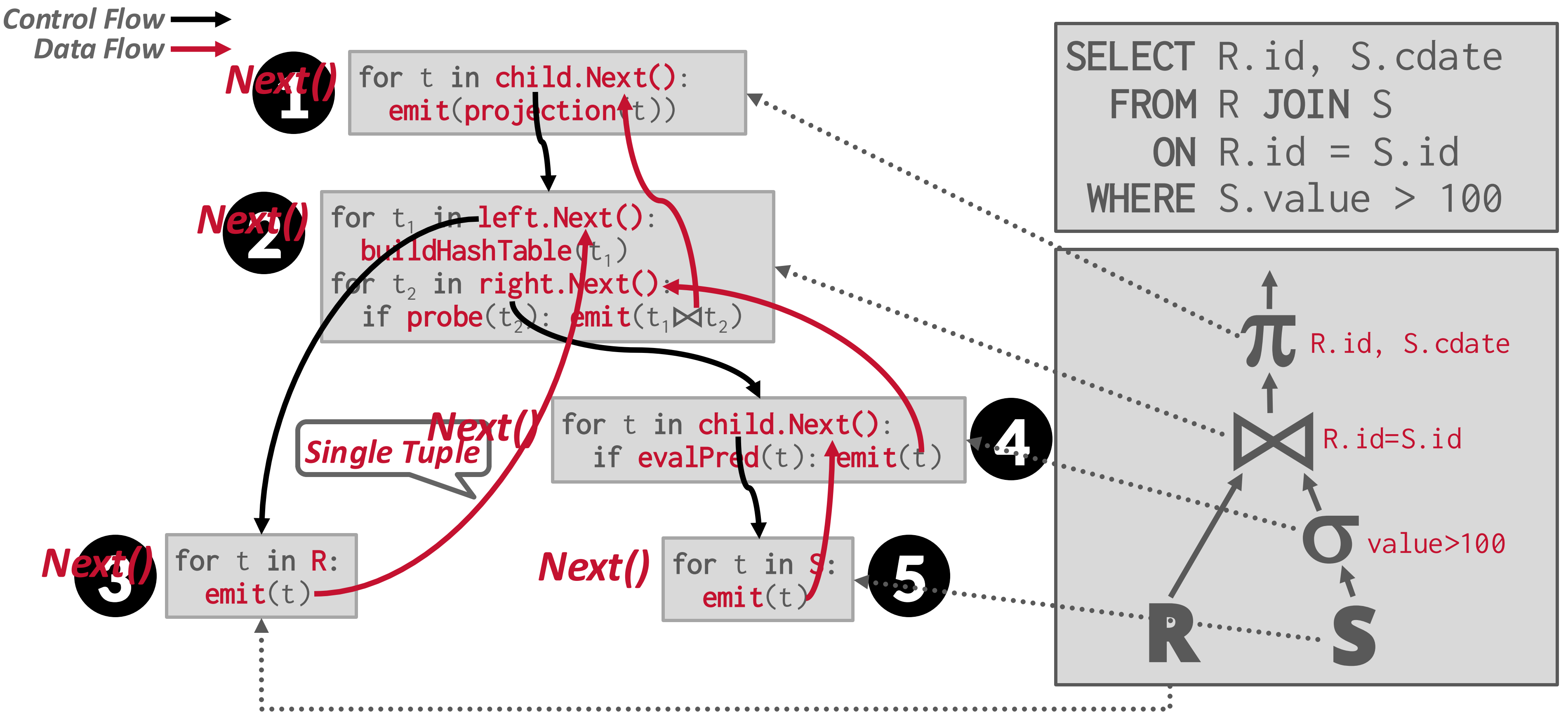
→ The operator implements a loop that calls `Next()` on its children to retrieve their tuples and then process them.

Each operator implementation also has `Open()` and `Close()` functions.

→ Analogous to constructors/destructors, but for operators.

Also called **Volcano** or **Pipeline** Model.

Iterator Model



```
for t in child.Next():
  emit(projection(t))
```

```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

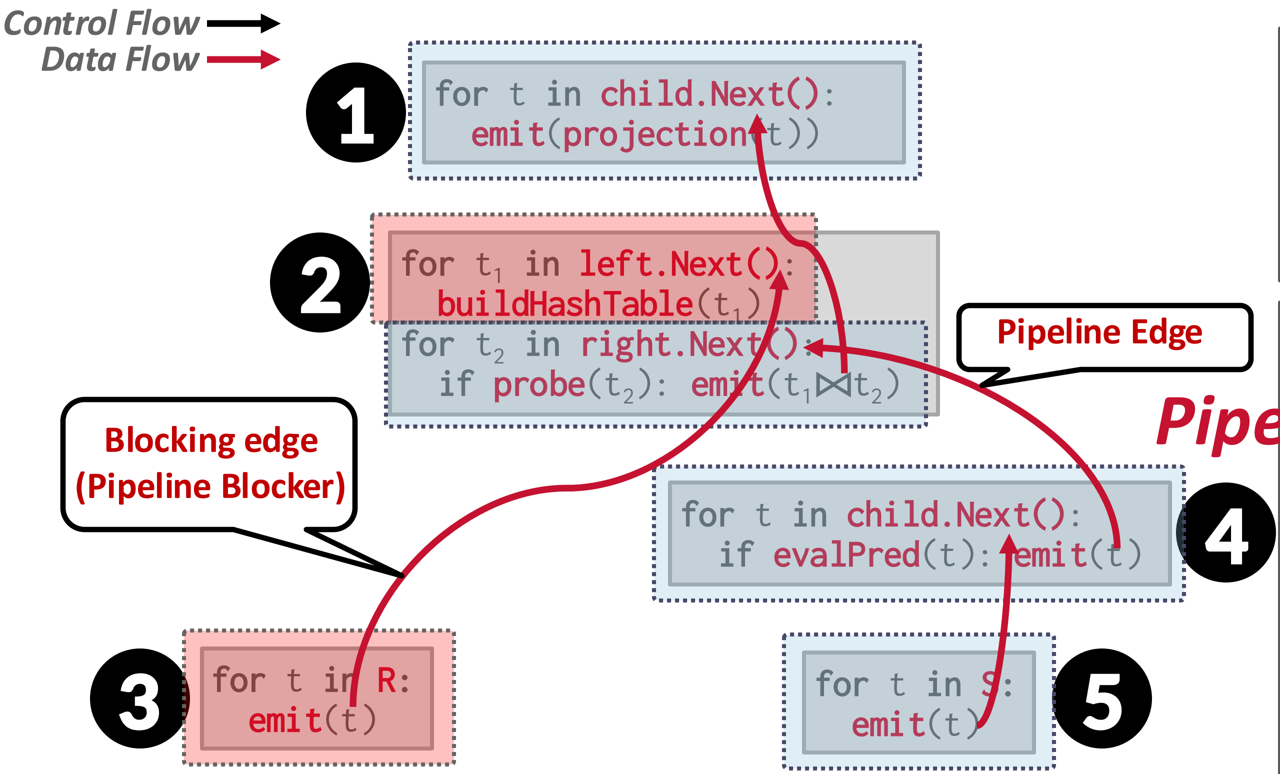
```
for t in R:
  emit(t)
```

```
for t in child.Next():
  if evalPred(t): emit(t)
```

```
for t in S:
  emit(t)
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

Iterator Model



```
1
for t in child.Next():
  emit(projection(t))
```

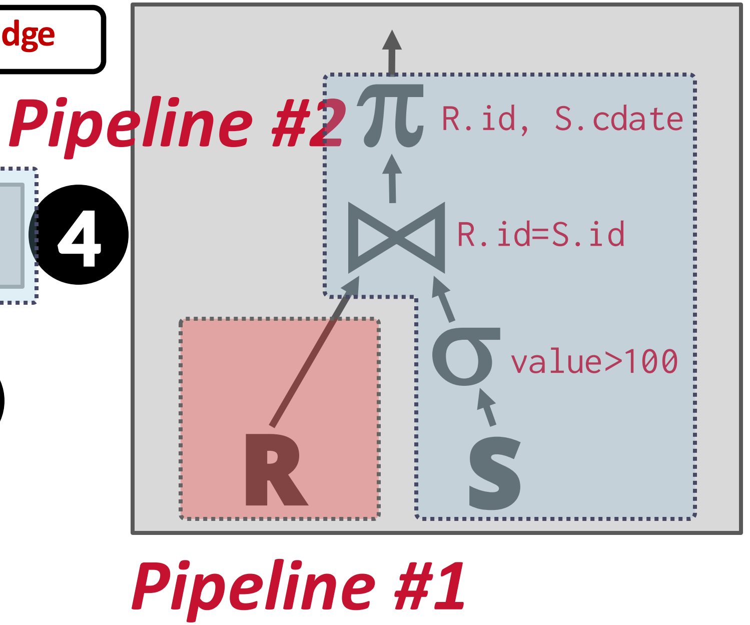
```
2
for t₁ in left.Next():
  buildHashTable(t₁)
for t₂ in right.Next():
  if probe(t₂): emit(t₁⋈t₂)
```

```
3
for t in R:
  emit(t)
```

```
4
for t in child.Next():
  if evalPred(t): emit(t)
```

```
5
for t in S:
  emit(t)
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Iterator Model

The Iterator model is used in almost every DBMS.

→ Easy to implement / debug.

→ Output control works easily with this approach.

Benefits of **<u>pipelining</u>**

→ Within pipeline, never spill intermediate results to disk

→ Start returning results sooner

→ Secret reason #3: pipeline parallelism (next class)

Materialization Model

Each operator processes its input all at once and then emits its output all at once.

→ The operator "materializes" its output as a single result.

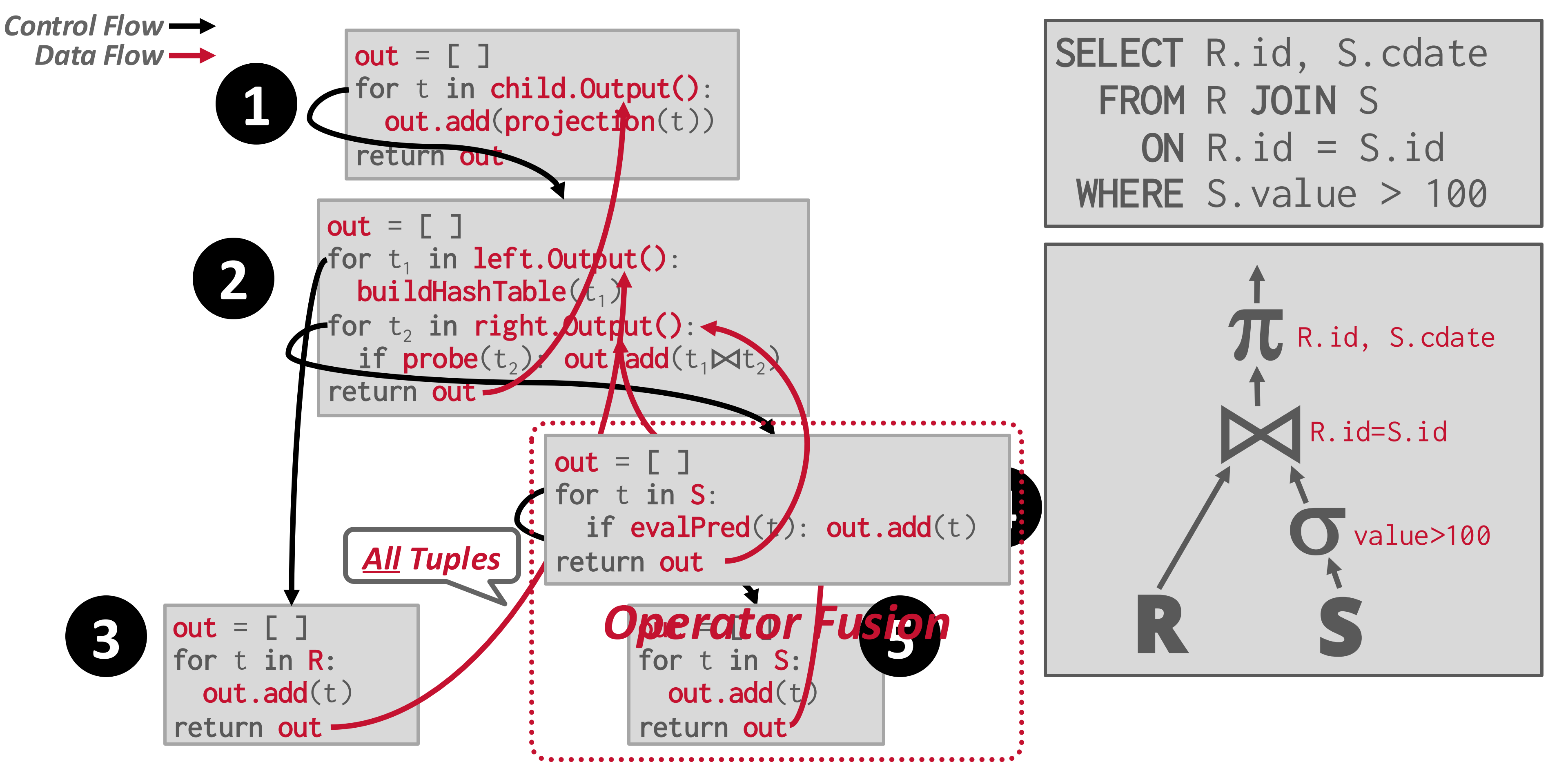
→ The DBMS can push down hints (e.g., `LIMIT`) to avoid scanning too many tuples.

→ Can send either a materialized row or a single column.

The output can be either whole tuples (NSM) or subsets of columns (DSM).

→ Originally developed by MonetDB in the 1990s to process entire columns at a time instead of single tuples.

Materialization Model



```
out = [ ]
for t in child.Output():
  out.add(projection(t))
return out
```

```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

```
out = [ ]
for t in R:
  out.add(t)
return out
```

```
out = [ ]
for t in S:
  if evalPred(t): out.add(t)
return out
```

```
out = [ ]
for t in S:
  out.add(t)
return out
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

Materialization Model

See your entire input set, pick the best algorithm

Better for OLTP workloads because queries only access a small number of tuples at a time.
→ Lower execution / coordination overhead.
→ Fewer function calls.

Not ideal for OLAP queries with large intermediate results, more optimizations needed

Vectorization Model

Like the Iterator Model where each operator implements a `Next()` function, but…

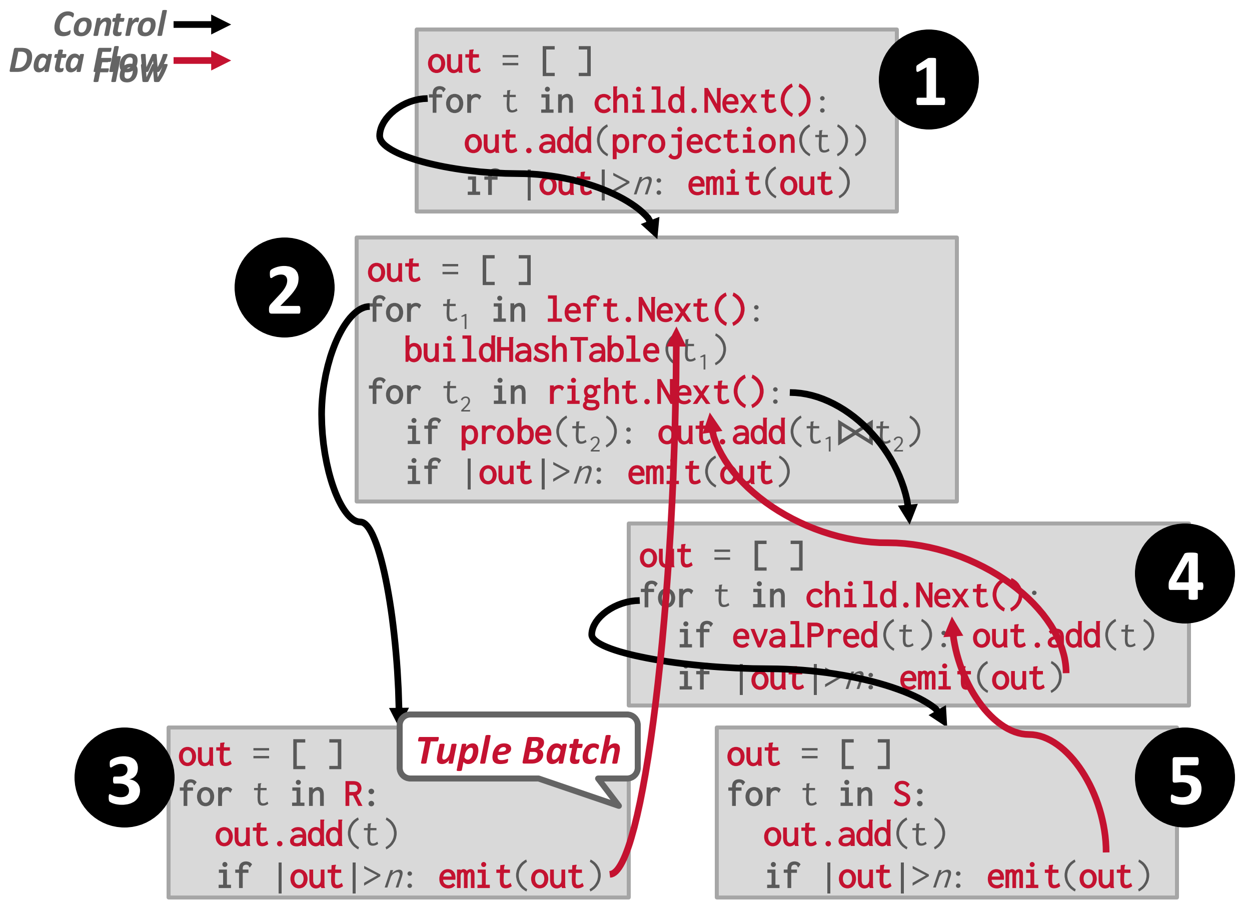
Each operator emits a **<u>batch</u>** of tuples instead of a single tuple.

→ The operator's internal loop processes multiple tuples at a time.

→ The size of the batch can vary based on hardware or query properties.

→ Each batch will contain one or more columns each their own null bitmaps.

Vectorization Model



```
out = [ ]
for t in child.Next():
  out.add(projection(t))
  if |out|>n: emit(out)
```

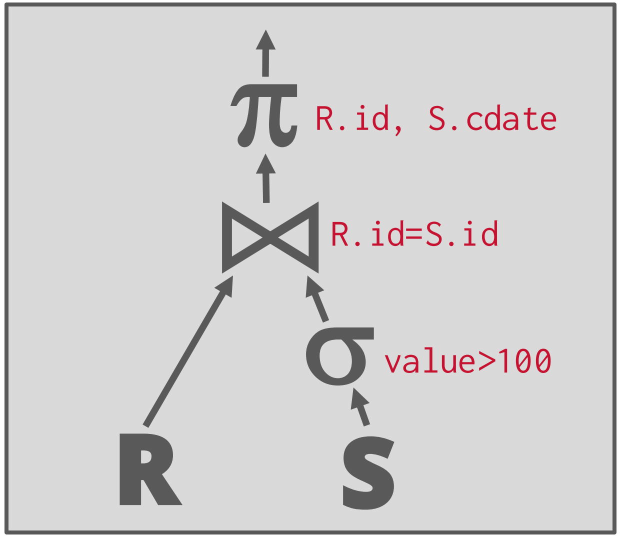
```
out = [ ]
for t₁ in left.Next():
  buildHashTable(t₁)
for t₂ in right.Next():
  if probe(t₂): out.add(t₁⋈t₂)
  if |out|>n: emit(out)
```

```
out = [ ]
for t in R:
  out.add(t)
  if |out|>n: emit(out)
```

```
out = [ ]
for t in child.Next():
  if evalPred(t): out.add(t)
  if |out|>n: emit(out)
```

```
out = [ ]
for t in S:
  out.add(t)
  if |out|>n: emit(out)
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Vectorization Model

Ideal for OLAP queries because it greatly reduces the number of invocations per operator.

Allows an out-of-order CPU to efficiently execute operators over batches of tuples.

→ Operators perform work in tight for-loops over arrays, which compilers know how to optimize / vectorize.
→ No data or control dependencies.
→ Hot instruction cache.

Observation

In the previous examples, the DBMS starts executing a query by invoking `Next()` at the root of the query plan and ***pulling*** data up from leaf operators.

This is the how most DBMSs implement their execution engine.

Plan Processing Direction

Approach #1: Top-to-Bottom (Pull)

→ Start with the root and "pull" data up from its children.

→ Tuples are always passed between operators using function calls (unless it's a pipeline breaker).

Approach #2: Bottom-to-Top (Push)

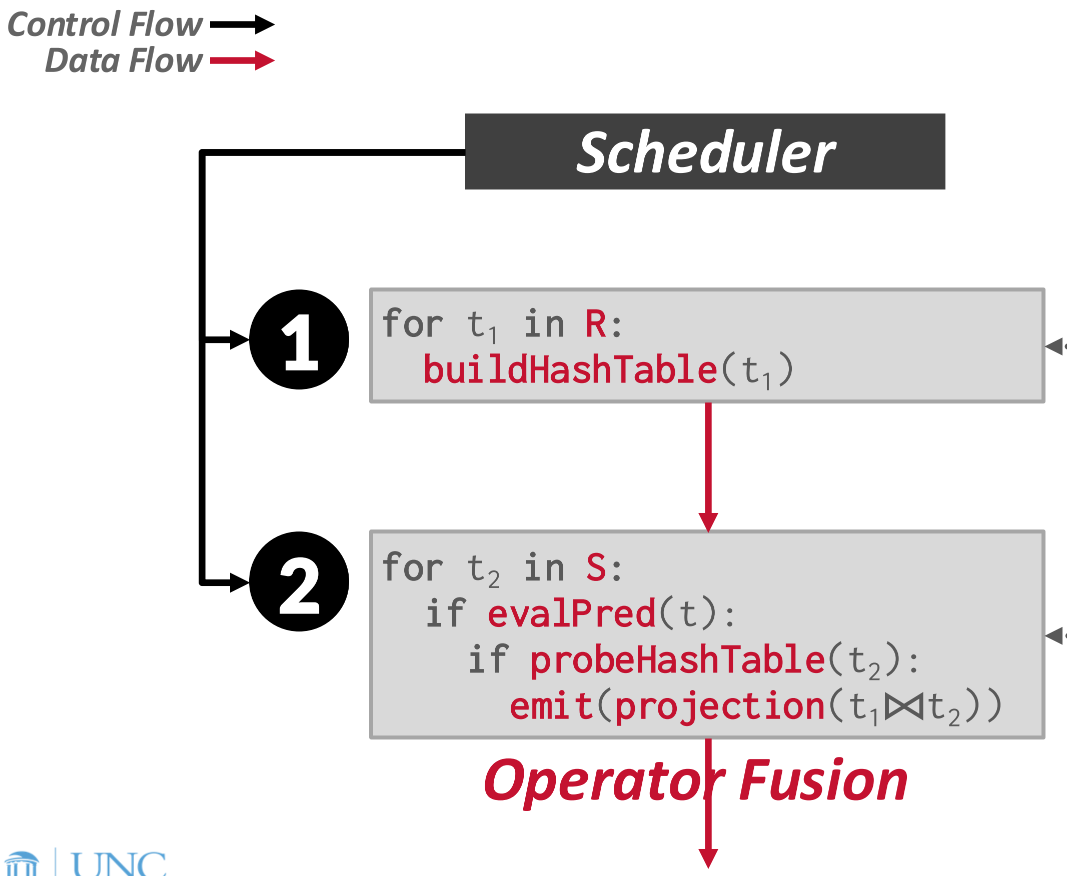
→ Start with leaf nodes and "push" data to their parents.

→ "fuse" operators together within a for-loop to minimize intermediate copies / function calls

→ Easier to orchestrate parallelism

snowflake DuckDB HyPer

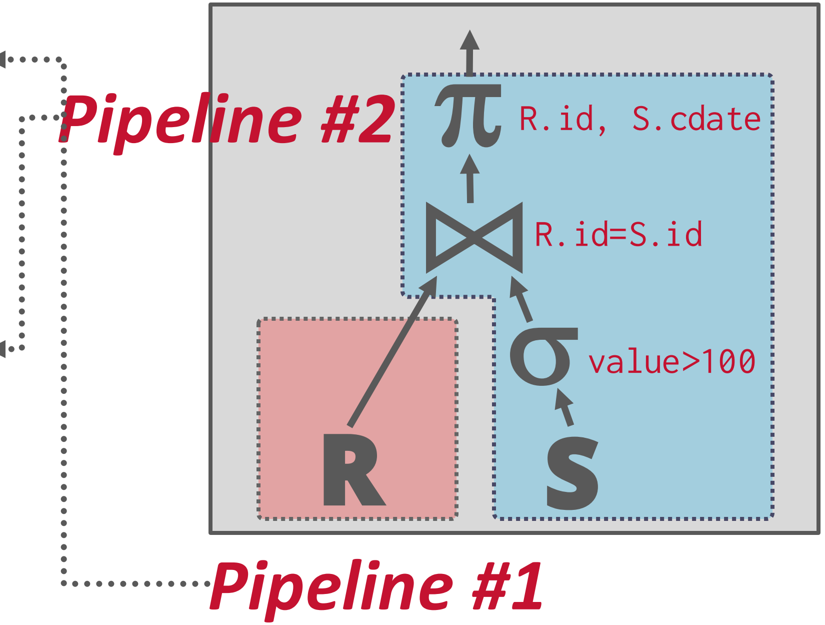
Push-based Iterator Model



```
for t₁ in R:
  buildHashTable(t₁)
```

```
for t₂ in S:
  if evalPred(t):
    if probeHashTable(t₂):
      emit(projection(t₁⋈t₂))
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Pu ed Iterator Model

Control Flo

D

Performance Evaluation 153 (2022) 102246

Contents lists available at ScienceDirect

Performance Evaluation

journal homepage: www.elsevier.com/locate/peva

ELSEVIER

Check for updates

The case for phase-aware scheduling of parallelizable jobs

Benjamin Berg [*], Justin Whitehouse, Benjamin Moseley, Weina Wang, Mor Harchol-Balter

Carnegie Mellon University, United States of America

ARTICLE INFO

Article history:
Available online 5 October 2021

Keywords:
Performance modeling
Parallel scheduling
Server allocation
Databases

ABSTRACT

Parallelizable jobs typically consist of multiple phases of computation, where the job is more parallelizable in some phases and less parallelizable in others. For example, in a database, a query may consist of a highly parallelizable table scan, followed by a less parallelizable table join. In the past, this phase-varying parallelizability was summarized by a single sub-linear speedup curve that measures a job's average parallelizability over its entire lifetime. Today, however, modern systems have fine-grained knowledge of the exact phase each job is in at every moment in time. Unfortunately, these systems do not fully leverage this real-time feedback when scheduling parallelizable jobs. Theory has failed to produce practical phase-aware scheduling policies, and thus scheduling in current systems is largely heuristic.

A phase-aware scheduling policy must decide, given its knowledge of each job's current phase, how many servers or cores to allocate to each job in the system at every moment in time. This paper provides the first stochastic model of a system processing parallelizable jobs composed of phases. Using our model, we derive an optimal phase-aware scheduling policy that minimizes the mean response time across jobs. Our provably optimal policy, Inelastic-First (IF), gives strict priority to jobs that are currently in less parallelizable phases. We validate our theoretical results using a simulation of a database running queries from the Star Schema Benchmark. We compare IF to a range of policies from both systems and theory, and show that IF reduces mean response time by up to a factor of 3.

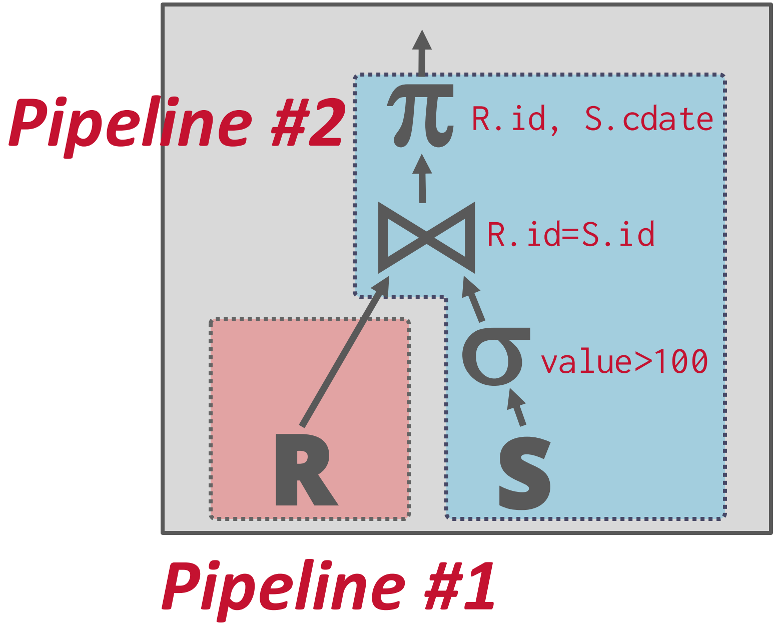
1. Introduction

Parallelizable workloads are ubiquitous and appear across a diverse array of modern computer systems. Data centers, supercomputers, machine learning clusters, distributed computing frameworks, and databases all process jobs designed to be parallelized across many servers or cores. Unlike the jobs in more classical models, such as the M/G/k queue, that each run on a single server, parallelizable jobs are capable of running on multiple servers simultaneously. When a job is parallelized across additional servers or cores, the job receives a speedup and can be completed more quickly.

When scheduling parallelizable jobs, a scheduling policy must decide how to best allocate servers or cores among the jobs in the system at every moment in time. This paper describes and analyzes scheduling policies for systems that process an online stream of incoming parallelizable jobs. Given a set of K servers, we will derive scheduling policies that minimize the mean response time across jobs — the average time from when a job arrives to the system until it is completed.

nding author.
bberg@cs.cmu.edu (B. Berg), jwhiteho@andrew.cmu.edu (J. Whitehouse), moseleyb@andrew.cmu.edu (B. Moseley), harchol@cs.cmu.edu (M. Harchol-Balter).

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Plan Processing Direction

Approach #1: Top-to-Bottom (Pull) ⬅ ***Most Common***

→ Easy to control output via `LIMIT`.
→ Parent operator blocks until its child returns with a tuple.
→ Additional overhead because operators' `Next()` functions are implemented as virtual functions.
→ Branching costs on each `Next()` invocation.

Approach #2: Bottom-to-Top (Push) ⬅ ***Rare***

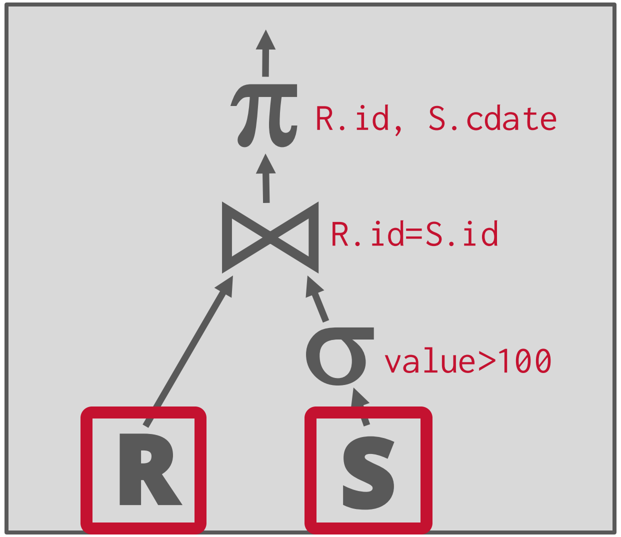
→ Allows for tighter control of caches/registers in pipelines.
→ May not have exact control of intermediate result sizes.
→ Difficult to implement some operators (Sort-Merge Join).

Access Methods

An **access method** is the way that the DBMS accesses the data stored in a table.
→ Not defined in relational algebra.

Three basic approaches:
→ Sequential Scan.
→ Index Scan (many variants).
→ Multi-Index Scan.

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



Sequential Scan

For each page in the table:

→ Retrieve it from the buffer pool manager.

→ Iterate over each tuple and check whether to include it.

The DBMS maintains an internal cursor that tracks the last page / slot it examined.

```
for page in table.pages:
  for t in page.tuples:
    if evalPred(t):
      // Do Something!
```

Sequential Scan: Optimizations

Lecture #5	Data Encoding / Compression
Lecture #06	Prefetching / Scan Sharing / Buffer Bypass
Lecture #08	Clustering / Sorting
Lecture #12	Late Materialization
	Materialized Views / Result Caching
	$\boxed{\text{Data Skipping}}$
	Code Specialization / Compilation
Lecture #14	Data Parallelization / Vectorization
Lecture #14	Task Parallelization / Multi-threading

Data Skipping

Approach #1: Approximate Queries (Lossy)

→ Execute queries on a sampled subset of the entire table to produce approximate results.

→ **Examples**: BlinkDB, Redshift, ComputeDB, XDB, Oracle, Snowflake, Google BigQuery, DataBricks

Approach #2: Zone Maps (Lossless)

→ Pre-compute columnar aggregations per page that allow the DBMS to check whether queries need to access it.

→ Trade-off between page size vs. filter efficacy.

→ **Examples**: Oracle, Vertica, SingleStore, Netezza, Snowflake, Google BigQuery

Zone Ma

Pre-computed aggregates for
in a page. DBMS checks the z
decide whether it wants to a

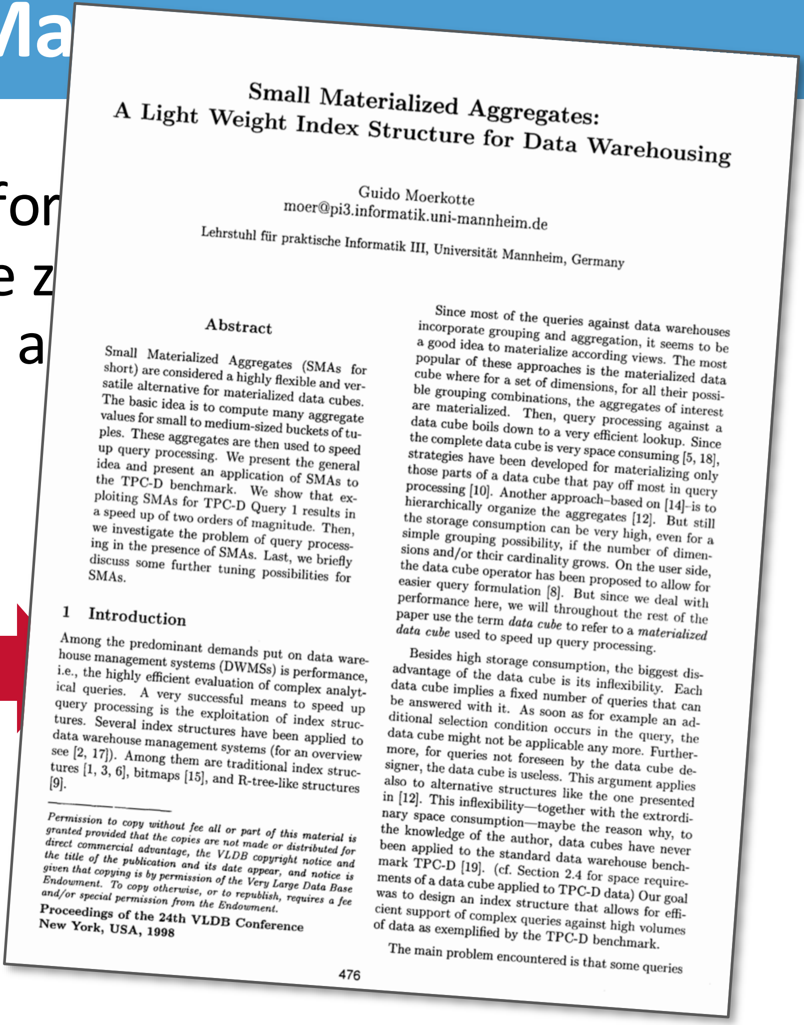
```
SELECT * FROM table
 WHERE val > 600
```

Parquet

Apache orc™

Original Data

val
100
200
300
400
400



Small Materialized Aggregates:
A Light Weight Index Structure for Data Warehousing

Guido Moerkotte
moer@pi3.informatik.uni-mannheim.de
Lehrstuhl für praktische Informatik III, Universität Mannheim, Germany

Abstract

Small Materialized Aggregates (SMAs for short) are considered a highly flexible and versatile alternative for materialized data cubes. The basic idea is to compute many aggregate values for small to medium-sized buckets of tuples. These aggregates are then used to speed up query processing. We present the general idea and present an application of SMAs to the TPC-D benchmark. We show that exploiting SMAs for TPC-D Query 1 results in a speed up of two orders of magnitude. Then, we investigate the problem of query processing in the presence of SMAs. Last, we briefly discuss some further tuning possibilities for SMAs.

1 Introduction

Among the predominant demands put on data warehouse management systems (DWMSs) is performance, i.e., the highly efficient evaluation of complex analytical queries. A very successful means to speed up query processing is the exploitation of index structures. Several index structures have been applied to data warehouse management systems (for an overview see [2, 17]). Among them are traditional index structures [1, 3, 6], bitmaps [15], and R-tree-like structures [9].

**Proceedings of the 24th VLDB Conference
New York, USA, 1998**

Since most of the queries against data warehouses incorporate grouping and aggregation, it seems to be a good idea to materialize according views. The most popular of these approaches is the materialized data cube where for a set of dimensions, for all their possible grouping combinations, the aggregates of interest are materialized. Then, query processing against a data cube boils down to a very efficient lookup. Since the complete data cube is very space consuming [5, 18], strategies have been developed for materializing only those parts of a data cube that pay off most in query processing [10]. Another approach–based on [14]–is to hierarchically organize the aggregates [12]. But still the storage consumption can be very high, even for a simple grouping possibility, if the number of dimensions and/or their cardinality grows. On the user side, the data cube operator has been proposed to allow for easier query formulation [8]. But since we deal with performance here, we will throughout the rest of the paper use the term *data cube* to refer to a *materialized data cube* used to speed up query processing.

Besides high storage consumption, the biggest disadvantage of the data cube is its inflexibility. Each data cube implies a fixed number of queries that can be answered with it. As soon as for example an additional selection condition occurs in the query, the data cube might not be applicable any more. Furthermore, for queries not foreseen by the data cube designer, the data cube is useless. This argument applies also to alternative structures like the one presented in [12]. This inflexibility—together with the extrordinary space consumption—maybe the reason why, to the knowledge of the author, data cubes have never been applied to the standard data warehouse benchmark TPC-D [19]. (cf. Section 2.4 for space requirements of a data cube applied to TPC-D data) Our goal was to design an index structure that allows for efficient support of complex queries against high volumes of data as exemplified by the TPC-D benchmark.

The main problem encountered is that some queries

Index Scan

The DBMS picks an index to find the tuples that the query needs.

Lecture #15

Which index to use depends on:

→ What attributes the index contains
→ What attributes the query references
→ The attribute's value domains
→ Predicate composition
→ Whether the index has unique or non-unique keys

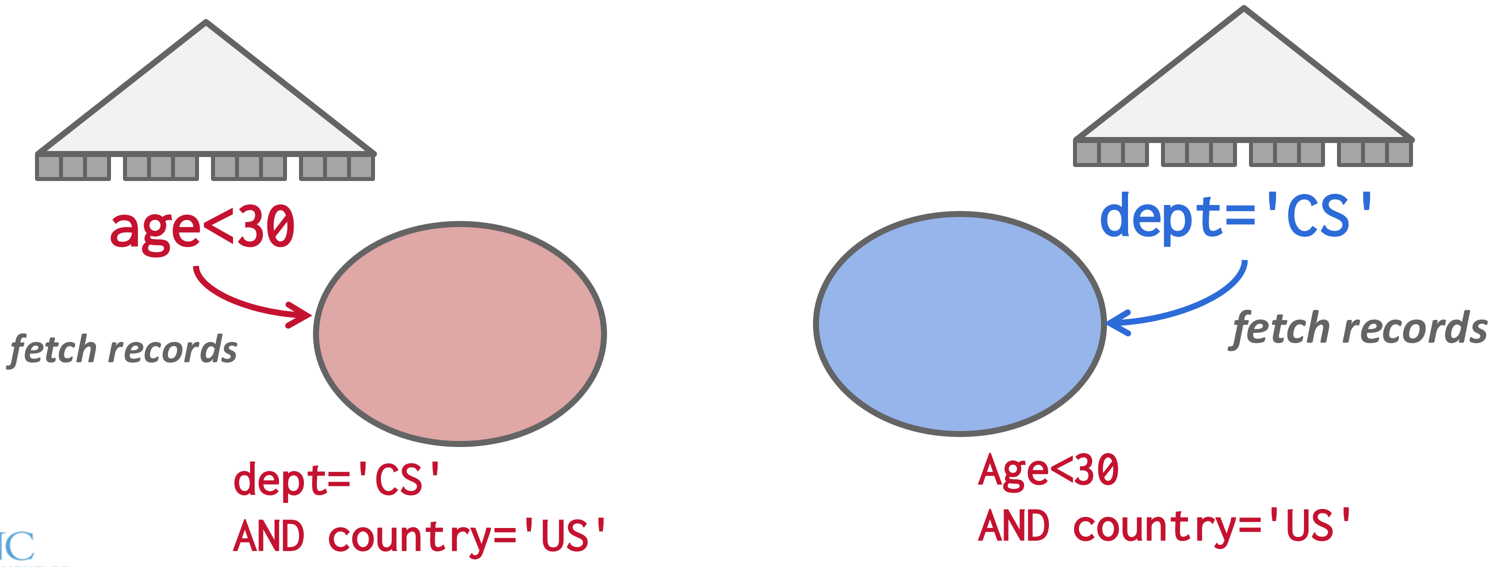
Index Scan

Suppose that we have a single table with 100 tuples and two indexes:

→ Index #1: age

→ Index #2: dept

```
SELECT * FROM students
 WHERE age < 30
   AND dept = 'CS'
   AND country = 'US'
```



Index Scan

Suppose that we have a single table with 100 tuples and two indexes:
→ Index #1: age
→ Index #2: dept

```
SELECT * FROM students
 WHERE age < 30
   AND dept = 'CS'
   AND country = 'US'
```

Scenario #1

There are 99 people under the age of 30 but only 2 people in the CS department.

Scenario #2

There are 99 people in the CS department but only 2 people under the age of 30.

Multi-index Scan

If there are multiple indexes available for a query, the DBMS does not have to pick only one:

→ Compute sets of Record IDs using each matching index.
→ Combine these sets based on the query's predicates (union vs. intersect).
→ Retrieve the records and apply any remaining predicates.

Examples:

→ DB2 Multi-Index Scan
→ PostgreSQL Bitmap Scan
→ MySQL Index Merge

Multi-index Scan

Given the following query on a database with an index #1 on age and an index #2 on dept:

→ We can retrieve the **Record IDs** satisfying age<30 using index #1.

→ Then retrieve the **Record IDs** satisfying dept='CS' using index #2.

→ Take their intersection.

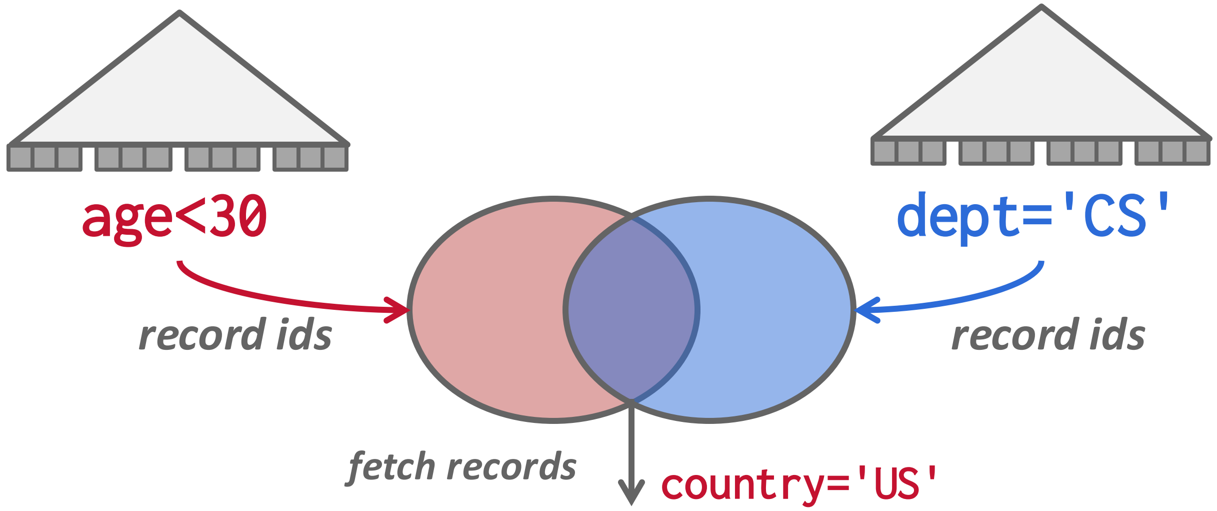
→ Retrieve records and check country='US'.

```
SELECT * FROM students
 WHERE age < 30
   AND dept = 'CS'
   AND country = 'US'
```

Multi-index Scan

Set intersection can be done efficiently with bitmaps or hash tables.

```
SELECT * FROM students
 WHERE age < 30
   AND dept = 'CS'
   AND country = 'US'
```



Modification Queries

Operators that modify the database (INSERT, UPDATE, DELETE) are responsible for modifying the target table and its indexes.

→ Constraint checks can either happen immediately inside of operator or deferred until later in query/transaction.

The output of these operators can either be Record IDs or tuple data (i.e., RETURNING).

Modification Queries

UPDATE/DELETE:

→ Child operators pass Record IDs for target tuples.
→ Must keep track of previously seen tuples.

INSERT:

→ **Choice #1**: Materialize tuples inside of the operator.
→ **Choice #2**: Operator inserts any tuple passed in from child operators.

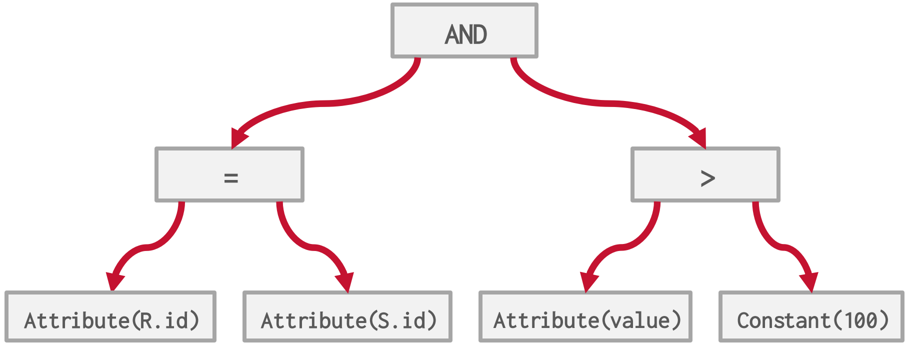
Expression Evaluation

The DBMS represents a WHERE clause as an **expression tree**.

The nodes in the tree represent different expression types:

→ Comparisons (=, <, >, !=)
→ Conjunction (AND), Disjunction (OR)
→ Arithmetic Operators (+, -, *, /, %)
→ Constant Values
→ Tuple Attribute References
→ Functions

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100;
```



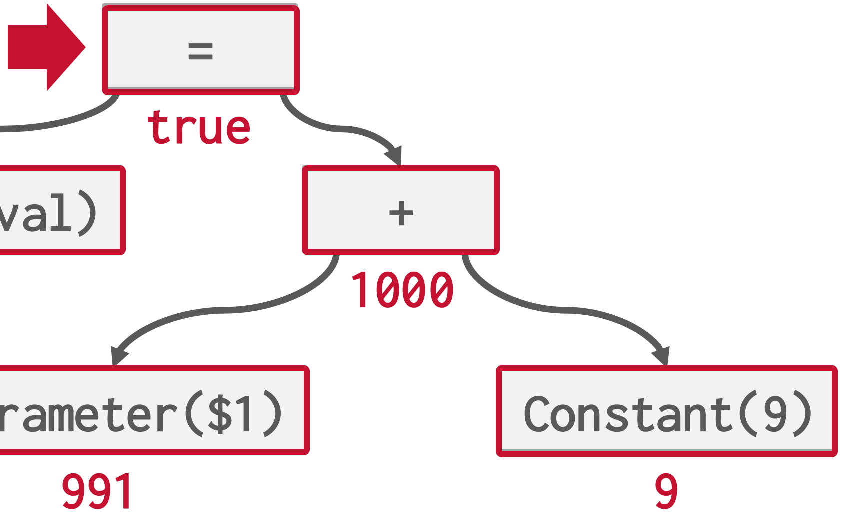
Expression Evaluation

```
PREPARE xxx AS
  SELECT * FROM S
   WHERE S.val = $1 + 9
```

```
EXECUTE xxx(991)
```

Execution Context

Current Tuple	Query Parameters	Table Schema
(123, 1000)	(int:991)	S→(int:id, int:val)

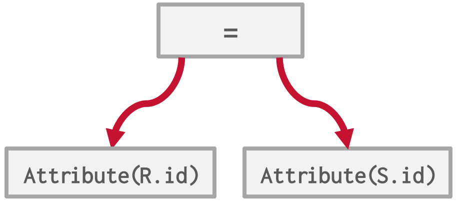


Expression Evaluation

```
PREPARE xxx AS
 SELECT * FROM S
  WHERE S.val = $1 + 9
```

```
EXECUTE xxx(991)
```

Is this a good idea?



Several Function calls (possible virtual)
Several pointer chases (depth-first traversal)
Dealing with schema / types / exec_ctx
Repeated work for every tuple

Fundamentally, what did we want to do?

```
cmp_eq $r1 $r2
je compare_true;
jmp compare_false;
```

We wanted ~2 instructions, we got hundreds or more!

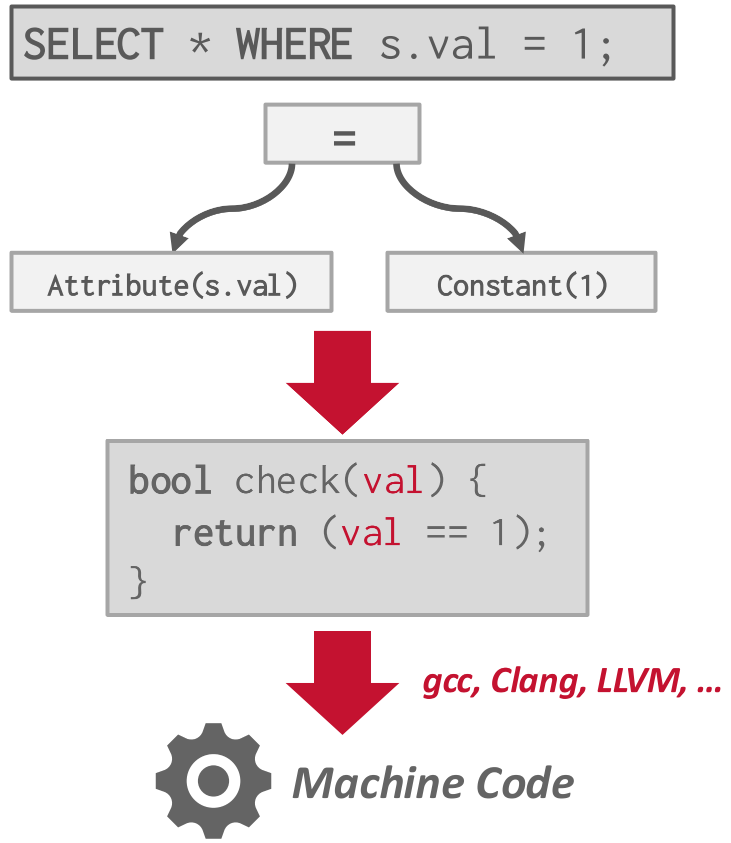
Expression / Query Compilation

Evaluating predicates by traversing a tree is terrible for the CPU.

→ The DBMS traverses the tree and for each node that it visits, it must figure out what the operator needs to do.

A better approach is to evaluate the expression directly.

An even better approach is to **vectorize** it evaluate a batch of tuples at the same time…



```
bool check(val) {
  return (val == 1);
}
```

CONCLUSION

The same query plan can be executed in multiple different ways.

(Most) DBMSs will want to use index scans as much as possible.

Expression trees are flexible but slow.
JIT compilation can (sometimes) speed them up.

NEXT CLASS

Parallel Query Execution