CS422
Database systems

Query Processing

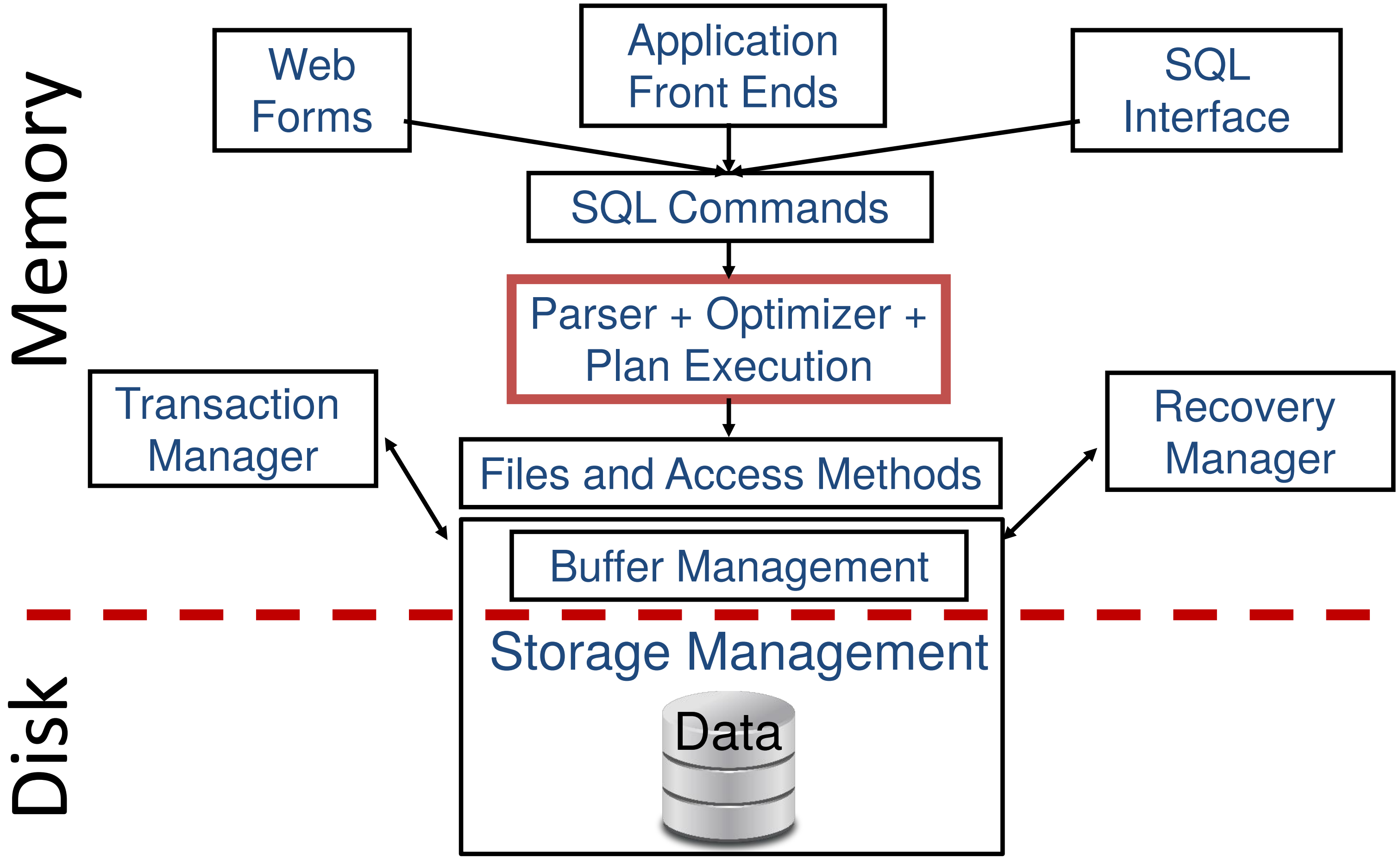
Data-Intensive Applications and Systems (DIAS) Laboratory
École Polytechnique Fédérale de Lausanne

"If we have data, let's look at data.
If all we have are opinions, let's go with mine"
– Jim Barksdale

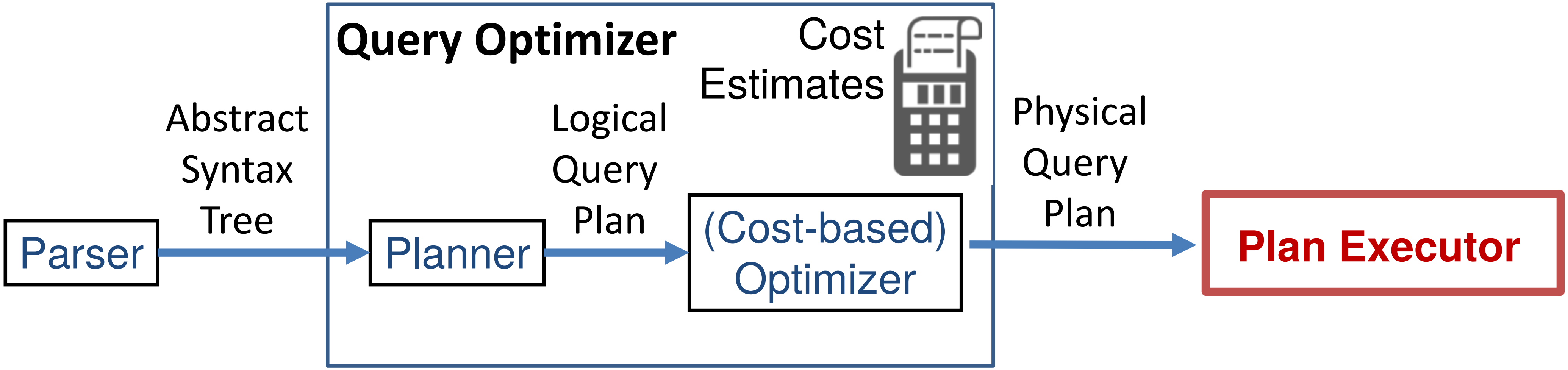
Some slides adapted from:

- Andy Pavlo
- CS-322

(Simplified) DBMS Architecture



Zoom in

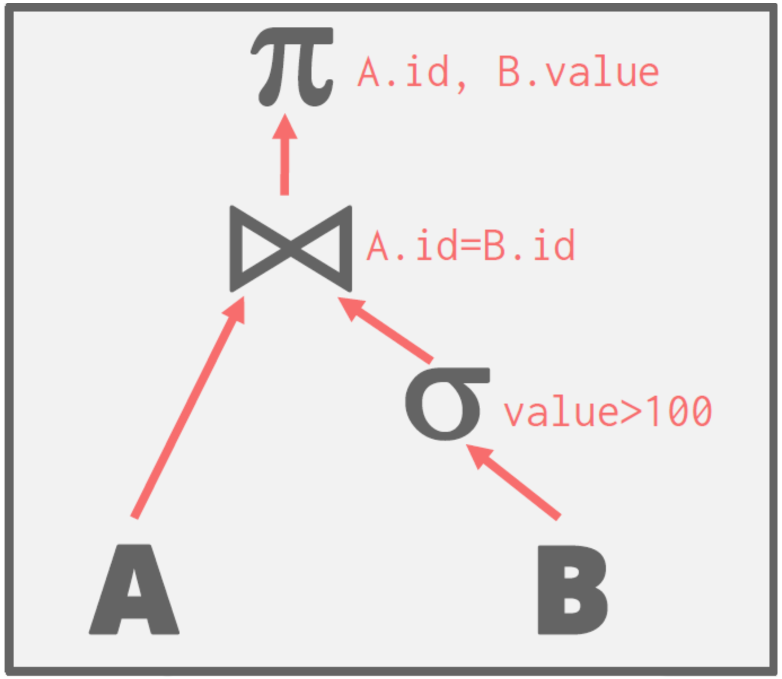


How the DBMS executes a query (plan)

Query Plan

- Operators are arranged in a tree.
- Data flows from leaves to root.
- Output of root = Query result.

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



Processing model

The ***processing model*** of a DBMS defines how the system executes a query plan.

- Different trade-offs for different workloads

- Extreme I: Tuple-at-a-time via the **iterator model**
- Extreme II: **Block-oriented model** (typically column-at-a-time)

Iterator Model

Each query plan operator implements a **next** function.

- On each invocation, the operator returns either a single tuple or a null 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.

Top-down plan processing.
Also called ***Volcano*** or ***Pipeline*** model.

Most popular method for (disk-oriented) DBMS

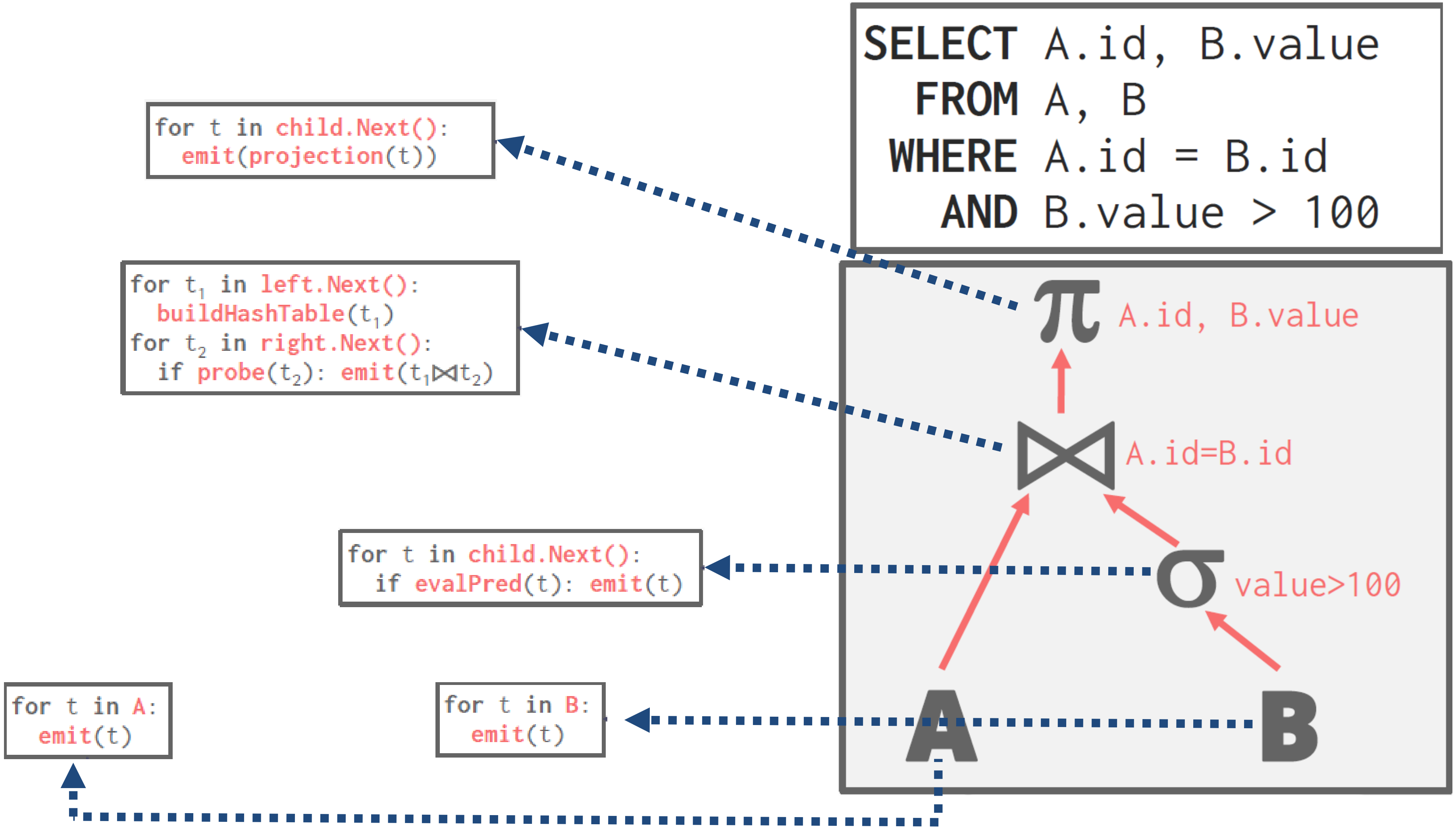
Iterator Model

- Used in almost every DBMS
- Allows for tuple **pipelining**

- Some operators will block until children emit all of their tuples.

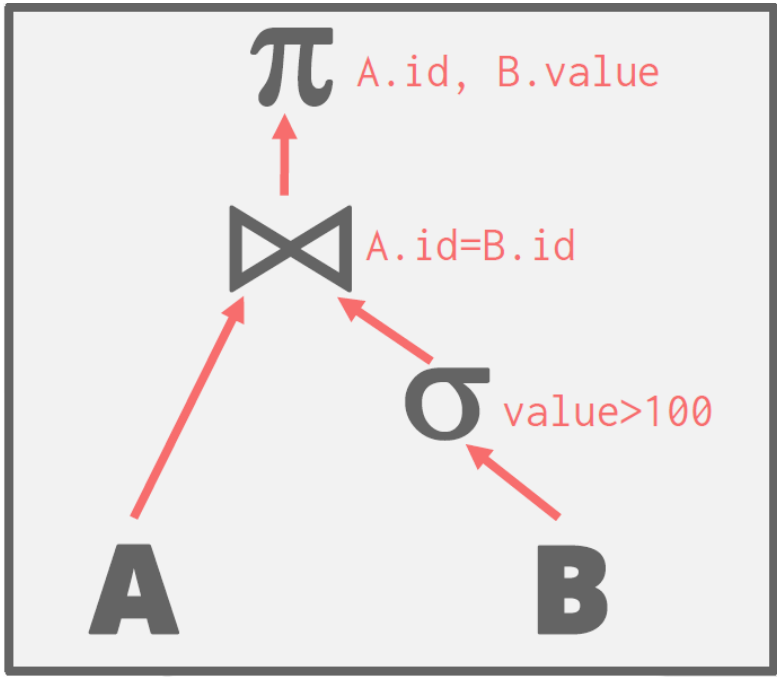
- Output control works easily
  - ***LIMIT*** operator

Iterator Model



Iterator Model

```sql
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



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 A:
  emit(t)
```

4

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

5

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

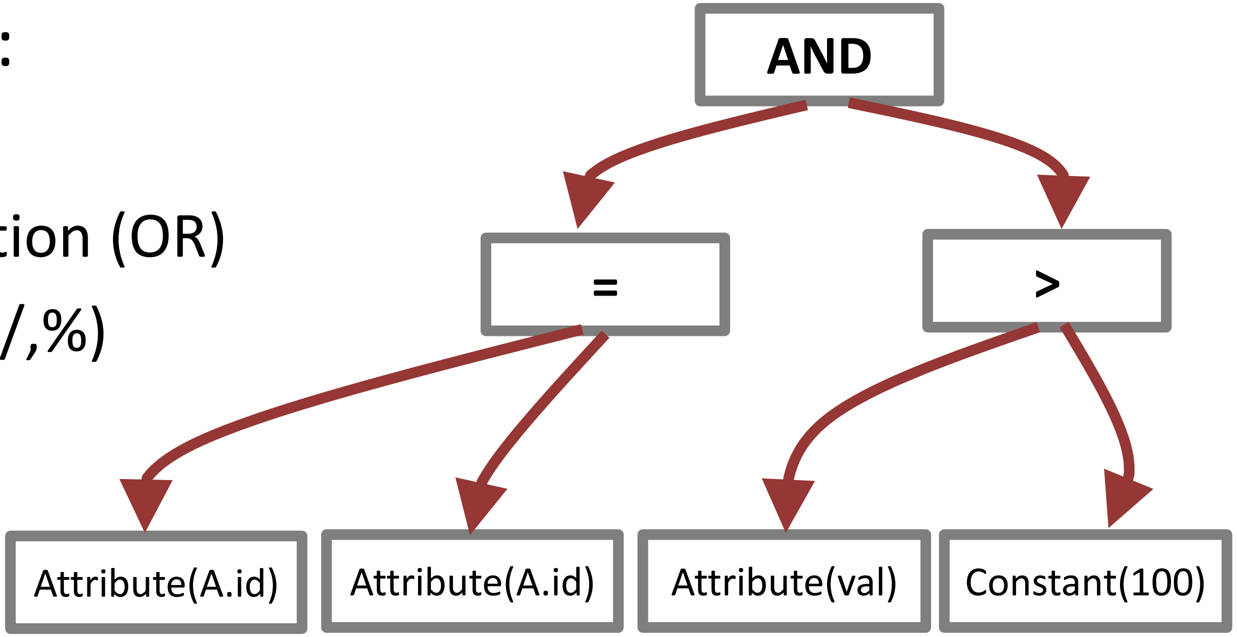
(Interpreted) Expression Evaluation

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```

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

Nodes in the tree represent different expression types:

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

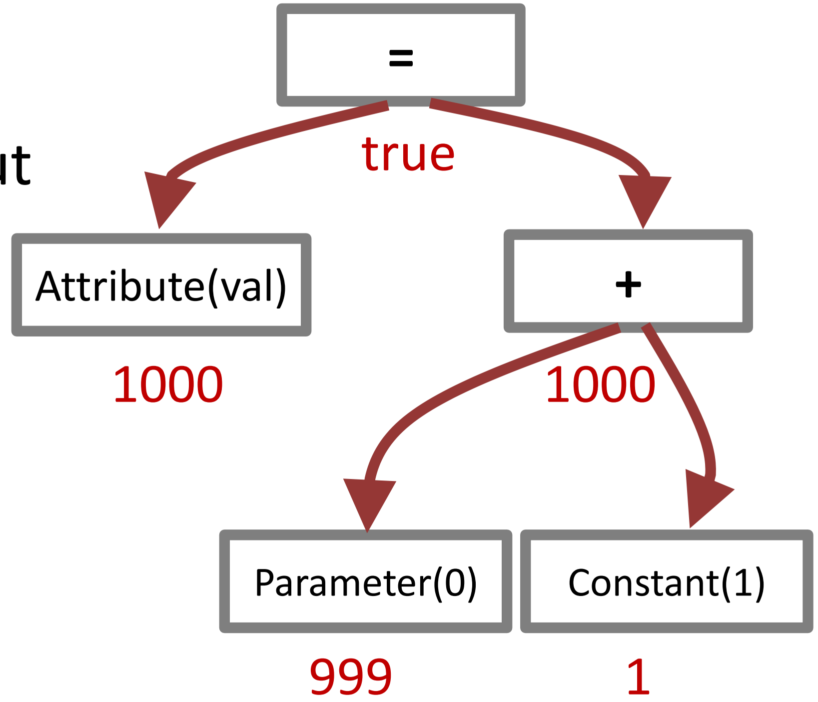


Expression Evaluation

```
SELECT * FROM B
WHERE B.val = ? + 1
```

Execution Context

Current Tuple	Query Parameters	Table Schema
(123, 1000)	(int:999)	B(int:id, int:val)



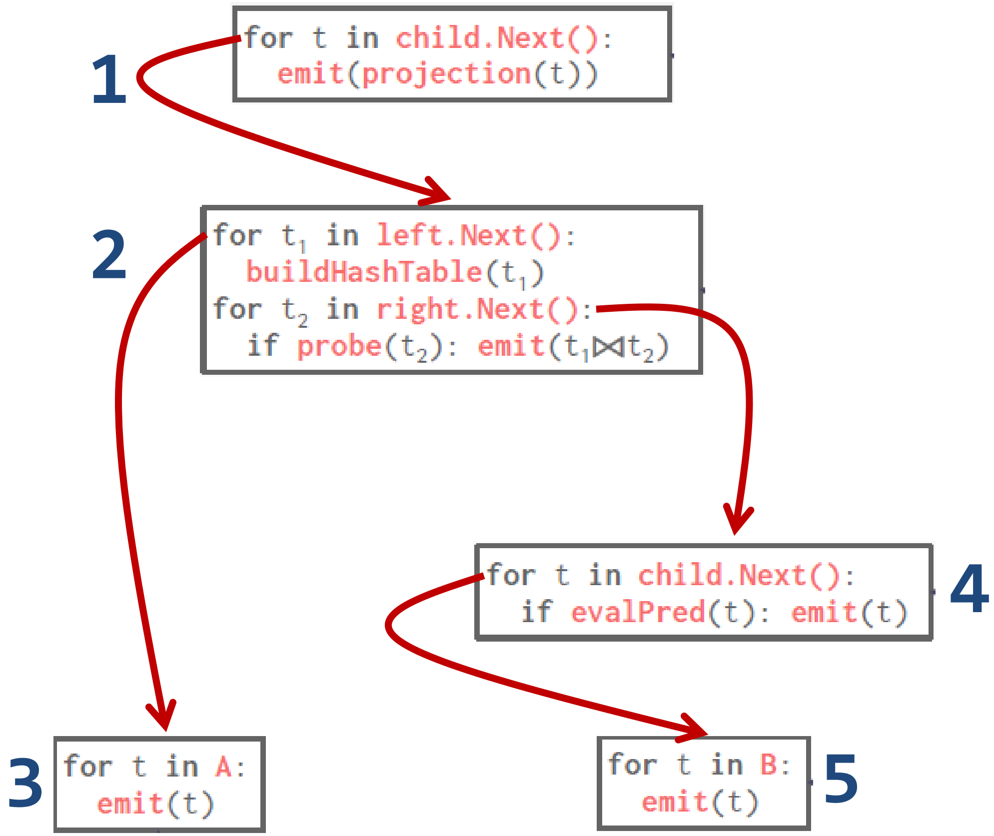
The DBMS traverses the tree.

For each node that it visits, it has to figure out what the operator needs to do.

This happens for every… single… tuple…

- Even for SELECT * FROM B
  WHERE 1 = 1
- It is **SLOW**

Interpreted, tuple-at-a-time processing



Many function calls

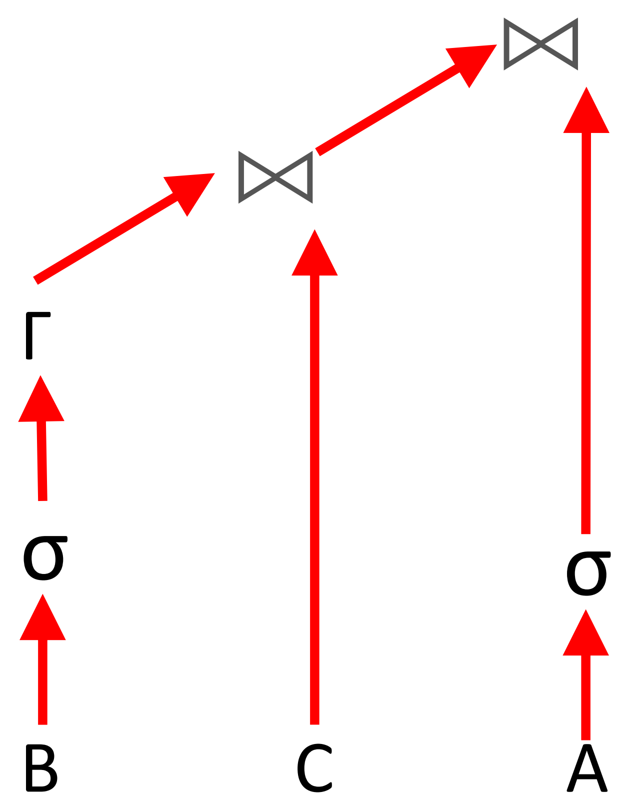
- Save/restore contents of CPU registers
- Force new instruction stream in the pipeline → bad for instruction cache

Generic code

- Has to cover every table, datatype, query

More complex queries, more problems

- Each getNext will invoke the getNext of child operator
- Many function calls for each tuple
- Many context switches
- Generic operators implementation: lot of code for type-checking & casting



Processing model

The ***processing model*** of a DBMS defines how the system executes a query plan.

- Different trade-offs for different workloads

- Extreme I: Tuple-at-a-time via the **iterator model**
- Extreme II: **Block-oriented model** (typically column-at-a-time)

Block-oriented (aka materialization) model

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

- The operator “materializes” its output as a single result.
- Bottom-up plan processing.

Block-oriented Model

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```

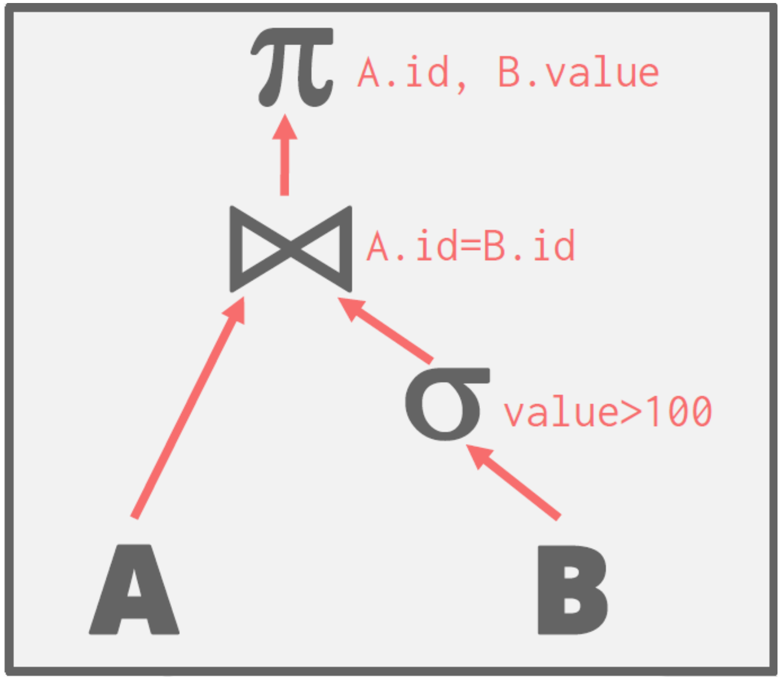
5
```
out = { }
for t in child.Output():
  out.add(projection(t))
```

4
```
out = { }
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
```

3
```
out = { }
for t in child.Output():
  if evalPred(t): out.add(t)
```

1
```
out = { }
for t in A:
  out.add(t)
```

2
```
out = { }
for t in B:
  out.add(t)
```

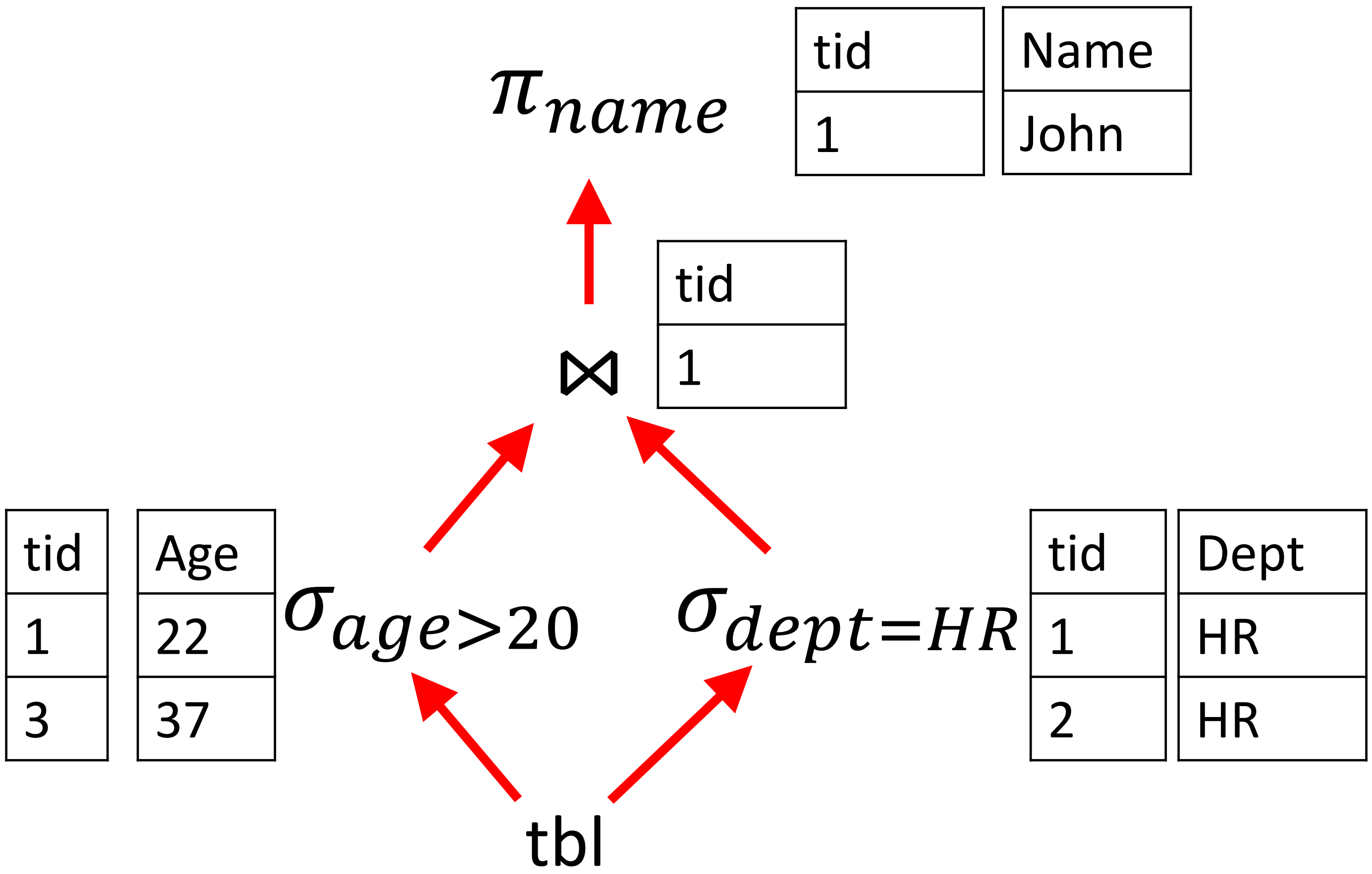


The (output) materialization problem – Naïve version

tbl

tid	Name	Age	Dept
1	John	22	HR
2	Jack	19	HR
3	Jane	37	IT

```
SELECT Name
FROM tbl
WHERE Age  > 20
      AND Dept = "HR"
```



The (output) materialization problem – version 2

tbl

tid	Name	Age	Dept
1	John	22	HR
2	Jack	19	HR
3	Jane	37	IT

```
SELECT Name
FROM tbl
WHERE Age  > 20
      AND Dept = “HR”
```

π_{name}

tid
1

Name
John

$\uparrow$

$\sigma_{dept=HR}$

tid
1

tid as extra filter to reduce output
Can we reduce it even more?

$\uparrow$

$\sigma_{age>20}$

tid
1
3

$\uparrow$

tbl

The (output) materialization problem – selection vector

```
SELECT Name
FROM tbl
WHERE Age  > 20
      AND Dept = "HR"
```

tbl

tid	Name	Age	Dept
1	John	22	HR
2	Jack	19	HR
3	Jane	37	IT

π_{name}

Name
John

$\sigma_{dept=HR}$

bitmap
1
0
0

$\sigma_{age>20}$

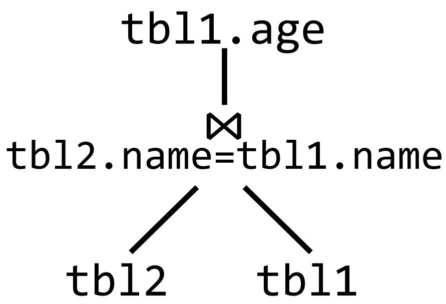
bitmap
1
0
1

tbl

- **Only materialize bitmap**
- **Perform calculations only for relevant tuples**

The (tuple) materialization problem

- When joining tables, columns can get shuffled
  => Cannot use virtual ids
  => Stitching causes random accesses



The order of tbl1.name entries can change after the join!!!

tid	Name
1	John
3	Jane
2	Jack

tid	Age
1	22
2	19
3	37

Option: Stitch columns before join

Option: Sort list of tids before projection

Option: Use order-preserving join algorithm (Not always applicable)

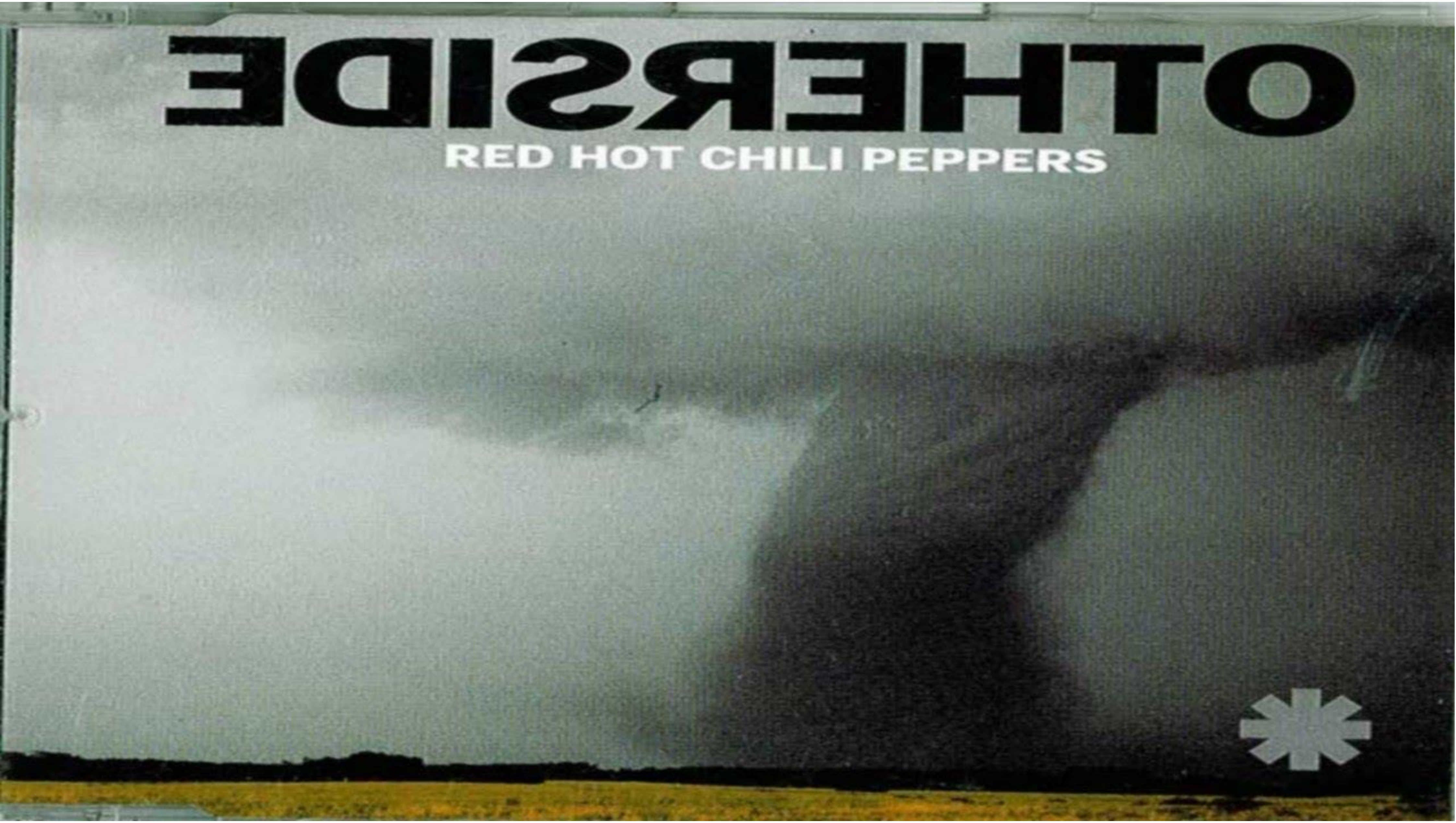
Block-oriented model

- ✓ No **next()** calls -> no per-tuple overhead
- ✓ Typically combined with columnar storage
  - Cache-friendly
  - SIMD-friendly
  - “Run same operation over consecutive data”
- ✓ Avoid interpretation when evaluating expressions (in most cases)
  - Typically use macros to produce 1000s of micro-operators (!!!)
    - selection_gt_int32(int *in, int pred, int *out)
    - selection_lt_int32(int *in, int pred, int *out)
    - …
- **Output materialization is costly** (in terms of memory bandwidth)

So far: The two extremes

Tuple-at-a-time execution

Column-at-a-time execution


OTHERSIDE
RED HOT CHILI PEPPERS

The beer analogy (by Marcin Zukowski): How to get 100 beers

Tuple-at-a-time execution:

- Go to the store
- Pick a beer bottle
- Pay at register
- Walk Home
- Put beer at bridge

Repeat till you have 100 beers

Many unnecessary steps

Column-at-a-time execution

- Go to the store
- Take 100 beers
- Pay at register
- Walk Home

100 beers not easy to carry

The middle ground: Vectorization model

- Like iterator model, each operator implements a **next** function
- Each operator emit a **vector** of tuples instead of a single tuple
  - Vector-at-a-time, aka **“Carry a crate of beers at a time”**!
  - The operator’s internal loop processes multiple tuples at a time.
  - Vector size varies based on hardware or query properties
    - General idea: Vector must fit in CPU cache

Vectorization model

1
```
out = { }
for t in child.Output():
  out.add(projection(t))
  if |out| > n:  emit(out)
```

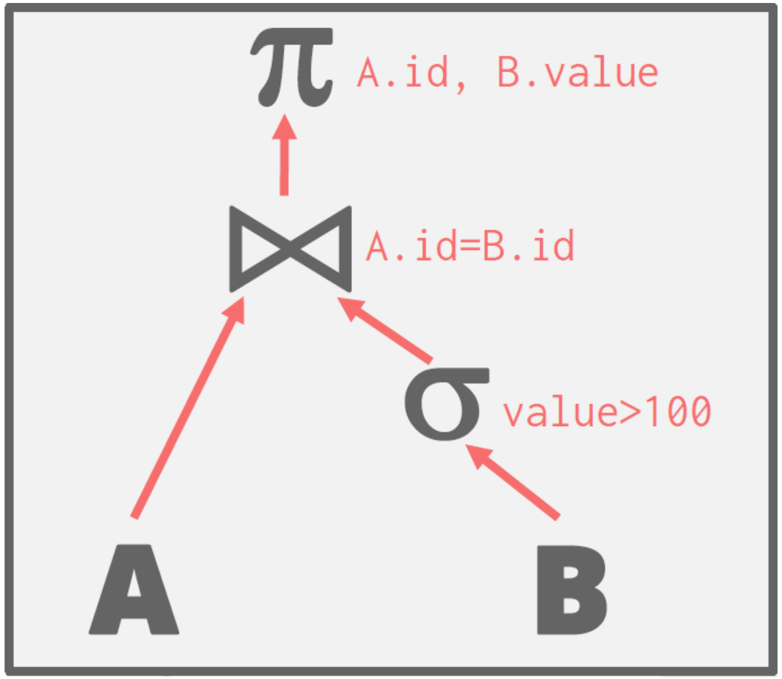
2
```
out = { }
for t1 in left.Output():
  buildHashTable(t1)
for t1 in right.Output():
  if probe(t2): out.add(t1 ⋈ t2)
  if |out| > n:   emit(out)
```

3
```
out = { }
for t in A:
  out.add(t)
  if |out| > n:   emit(out)
```

4
```
out = { }
for t1 in child.Output():
  if evalPred(t):   out.add(t)
  if |out| > n:   emit(out)
```

5
```
out = { }
for t in B:
  out.add(t)
  if |out| > n:   emit(out)
```

```sql
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



Vectorization model

Ideal for OLAP queries

- Greatly reduces the number of invocations per operator
- Allows for operators to use vectorized (SIMD) instructions to process batches of tuples

Execution of the physical plan

Two approaches

- Traditional approach: interpretation
- Code generation & compilation

Processing model

The ***processing model*** of a DBMS defines how the system executes a query plan.

– Different trade-offs for different workloads

- Extreme I: Tuple-at-a-time via the **iterator model**
- Query compilation
- Vectorization model
- Extreme II: **Block-oriented model** (typically column-at-a-time)

Remark from Microsoft Hekaton

After switching to an in-memory DBMS, the only way to increase throughput is to reduce the number of instructions executed.

- To go **10x** faster, the DBMS must execute **90%** fewer instructions
- To go **100x** faster, the DBMS must execute **99%** fewer instruction

The only way to achieve such a reduction in the number of instructions is through **code specialization.**

- Generate code that is specific to a particular task in the DBMS.
- (Currently, most code is written to be understandable)

Move from general to specialized code

- Any CPU intensive entity of a database can be natively compiled if they have a similar execution pattern on different inputs
  - Access Methods
  - Operator Execution
  - Predicate Evaluation
- Goal: Avoid runtime decisions!
  Decide once, when you see the query plan!

What do we know from the query plan?

- Attribute types

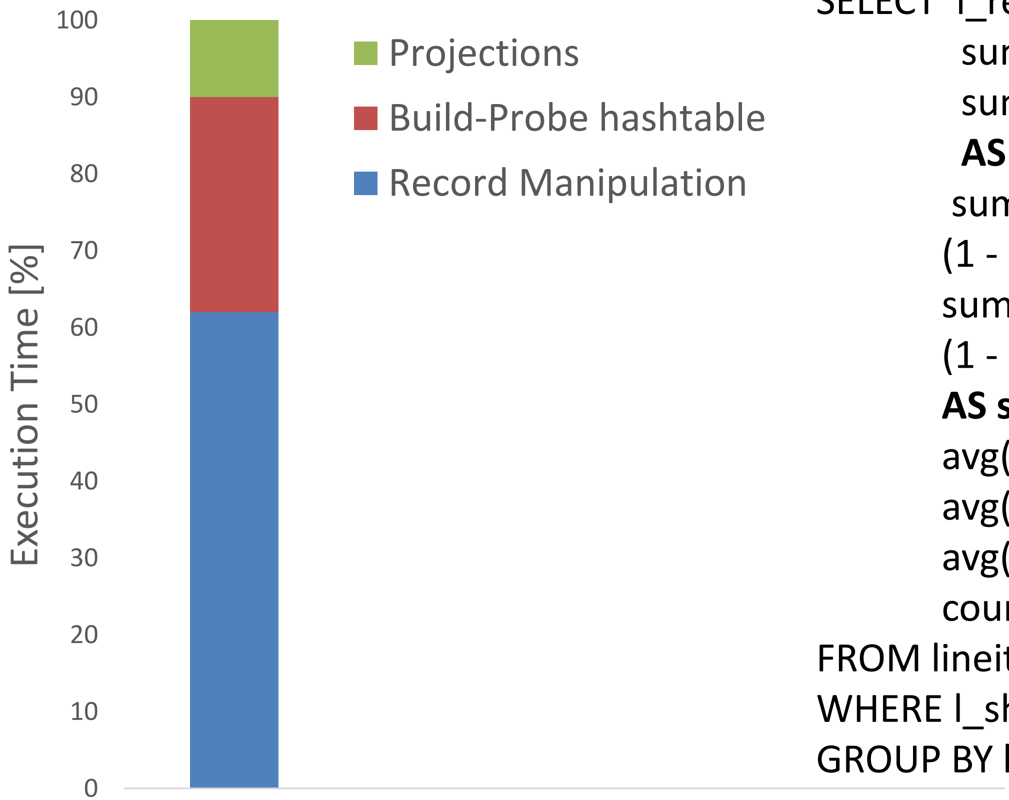
  => (Inline) pointer casting instead of
  data access (virtual) function calls

- Query predicate types

  => Can use primitive data comparisons

“Hard-code” this knowledge into the execution engine

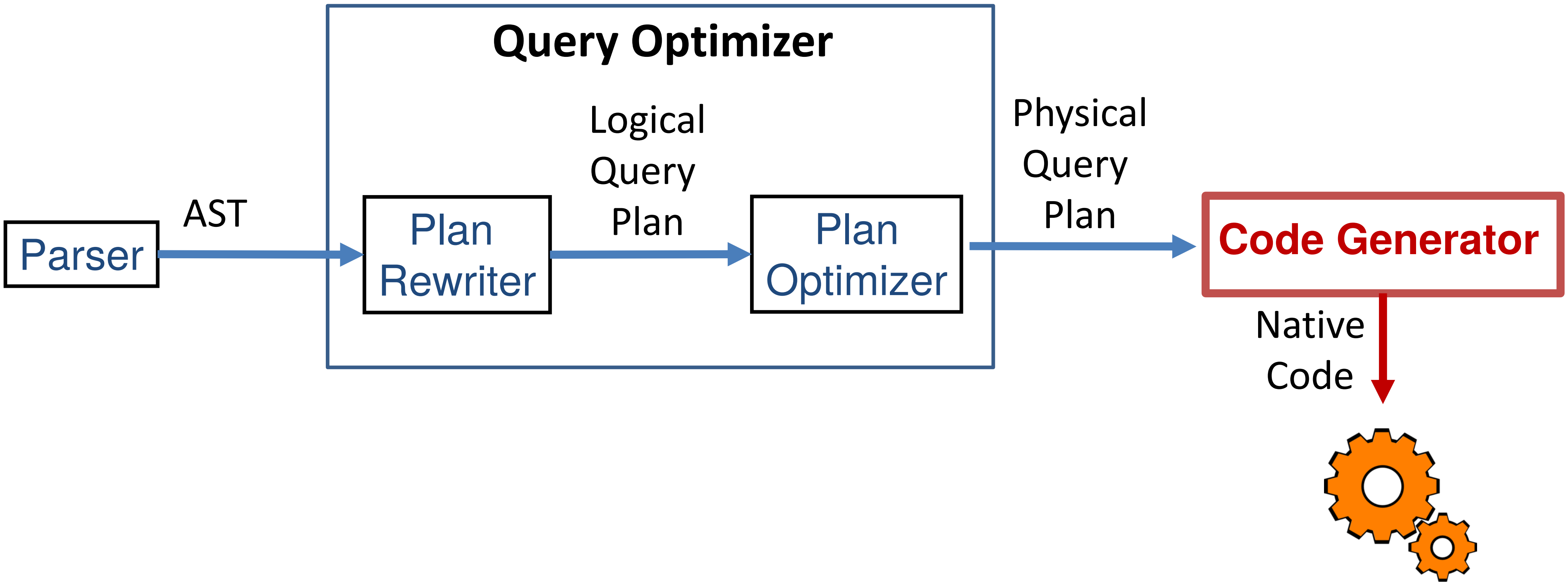
MySQL performance breakdown



SELECT l_returnflag, l_linestatus,
sum(l_quantity) **AS sum_qty**,
sum(l_extendedprice)
AS sum_base_price,
sum(l_extendedprice *
(1 - l_discount)) **AS sum_disc_price**,
sum(l_extendedprice *
(1 - l_discount) * (1 + l_tax))
AS sum_charge,
avg(l_quantity) **AS avg_qty**,
avg(l_extendedprice) **AS avg_price**,
avg(l_discount) **AS avg_disc**,
count(*) **AS count_order**
FROM lineitem
WHERE l_shipdate <= date '1998-09-02'
GROUP BY l_returnflag, l_linestatus

[Locking + Buffer Allocation costs minimal]

Query Compiler



Two approaches for code generation

Transpilation

- DBMS converts a relational query plan into C/C++ code
- Compile the produced code to generate native

JIT compilation

- Generate an intermediate representation (IR) of the query that can be quickly compiled into native code.

Transpilation use case: The HIQUE system

- HIQUE: Holistic Integrated QUery Engine
- For a given query plan, create a C program that implements that query's execution.
  → Bake in all the predicates and type conversions.
- Use an off-shelf compiler to convert the code into a shared object, link it to the DBMS process, and then invoke the exec function.

Operator templates

```
SELECT * FROM A WHERE A.val = ? + 1
```

Known at query compile time

Interpreted plan

for t **in range**(table.num_tuples):
  tuple = get_tuple(table, t)
if eval(predicate, tuple, params):
  emit(tuple)

1. Get schema in catalog for table
2. Calculate offset based on tuple size
3. Return pointer to tuple

1. Traverse predicate tree and pull values up
2. For tuple values, calculate the offset of the target attribute
3. Resolve datatype (switch / virtual call)
4. Return true/false

Templated plan

tuple_size = **###**
predicate_offset = **###**
parameter_value = **###**

for t **in range**(table.num_tuples):
  tuple = table.data + t ∗ tuple_size
  val = (tuple+predicate_offset)
  if (val == parameter_value + 1):
    emit(tuple)

Similar benefits for all other operators

```
SELECT * FROM A WHERE A.val = ? + 1
```

Interpreted plan

```
for t in range(table.num_tuples):
  tuple = get_tuple(table, t)
if eval(predicate, tuple, params):
  emit(tuple)
```

1. Get schema in catalog for table
2. Calculate offset based on tuple size
3. Return pointer to tuple

1. Traverse predicate tree and pull values up
2. For tuple values, calculate the offset of the target attribute
3. Perform casting
4. Return true/false

Templated plan

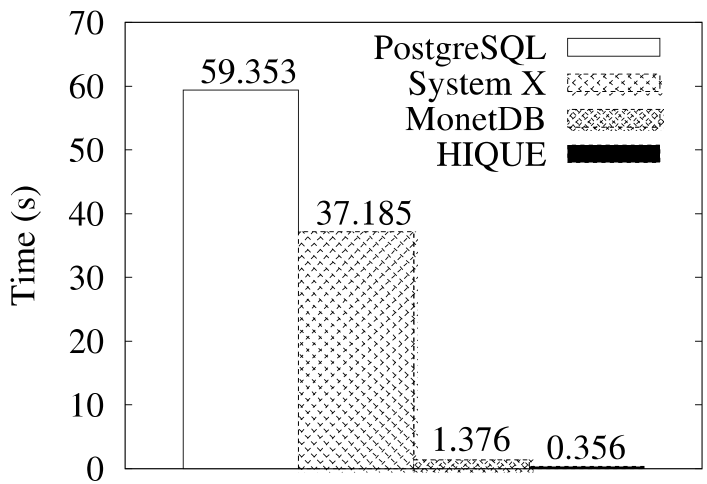
```
tuple_size = ###
predicate_offset = ###
parameter_value = ###

for t in range(table.num_tuples):
  tuple = table.data + t ∗ tuple_size
  val = (tuple+predicate_offset)
  if (val == parameter_value + 1):
    emit(tuple)
```

Predicate evaluation becomes **a single line**!

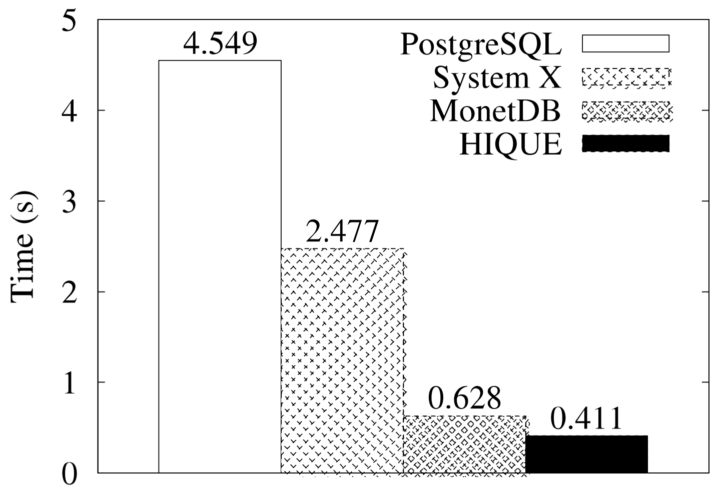
Integrating with the rest of the DBMS

- The generated query code can invoke any other function in the DBMS → no need to generate code for the whole DB!
- Re-use the same components as interpreted queries.
  - Concurrency control
  - Logging and checkpoints
  - Indexes

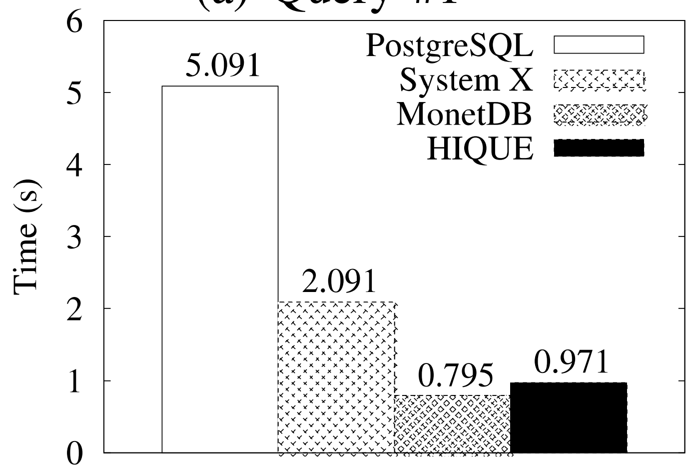
Indicative Performance



(a) Query #1



(b) Query #3



(c) Query #10

Up to 2 orders of magnitude improvement compared to interpreted DBs (PostgreSQL)

The catch

TPC-H	SQL processing (ms)			Compilation (ms)		C file sizes (bytes)	
Query	Parsing	Optimisation	Generation	with `-O0`	with `-O2`	Source	Shared library
1	21	1	1	121	274	17733	16858
3	11	1	2	160	403	33795	24941
5	11	1	2	201	578	43424	33088
10	15	1	4	213	619	50718	33510

Compilation takes time!

In practice, ~1 second is not a big issue for **OLAP** queries

- An OLAP query may take tens to hundreds of seconds
- How about OLTP queries?
- Hint: In OLTP, we know the typical queries → pre-compile and cache

HIQUE take-home message

- Reduce function calls
- Specialized code → avoid casting & type-checking, smaller code, promote cache reuse

BUT

- Compilation takes time
- How about complex plans?
- Sticks to the operator "legacy" abstraction

JIT compilation: The HyPer approach

- Generate code using LLVM
- LLVM: Collection of modular and reusable compiler and toolchain technologies.
- Core component is a low-level programming language (IR) that is similar to assembly.
- Not all of the DBMS components need to be written in LLVM IR.
  → LLVM code can make calls to C++ code.
- HyPer goal: "Keep a tuple in CPU registers as long as possible"
  - Push data through execution plan
  - **Blur operator boundaries**

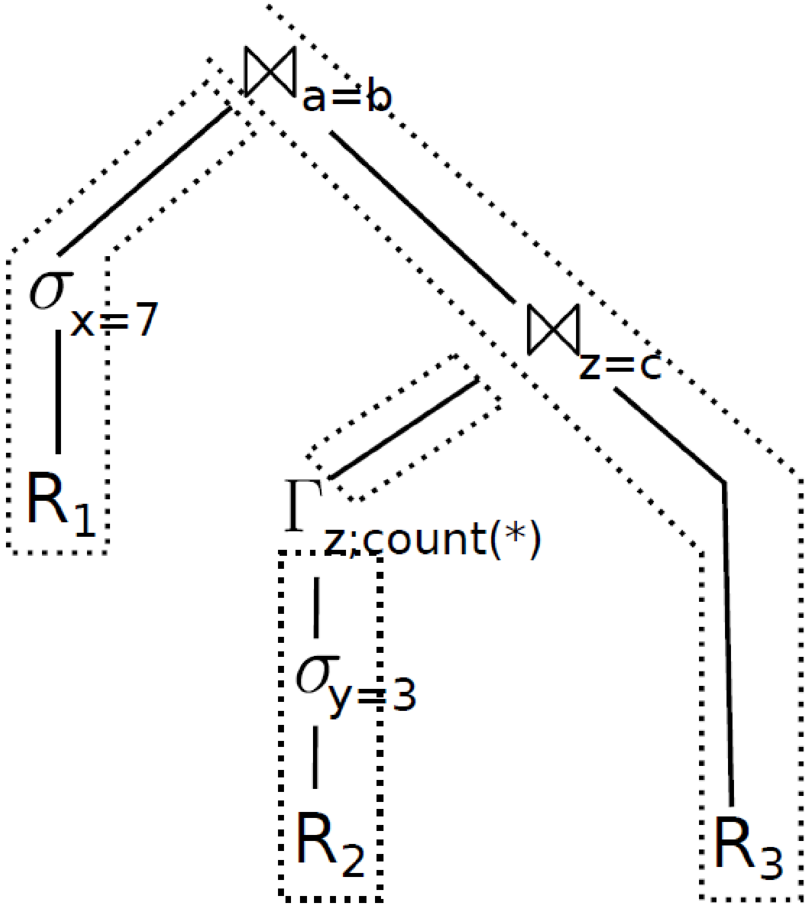
LLVM example: input and output

```
int mul_add(int x, int y, int z) {
  return x * y + z;
}
```

```
define i32 @mul_add(i32 %x, i32 %y, i32 %z)
{
  entry:
    %tmp = mul i32 %x, %y
    %tmp2 = add i32 %tmp, %z
    ret i32 %tmp2
}
```

- Produced code very close to assembly
- Compilation very fast (tens of milliseconds!)

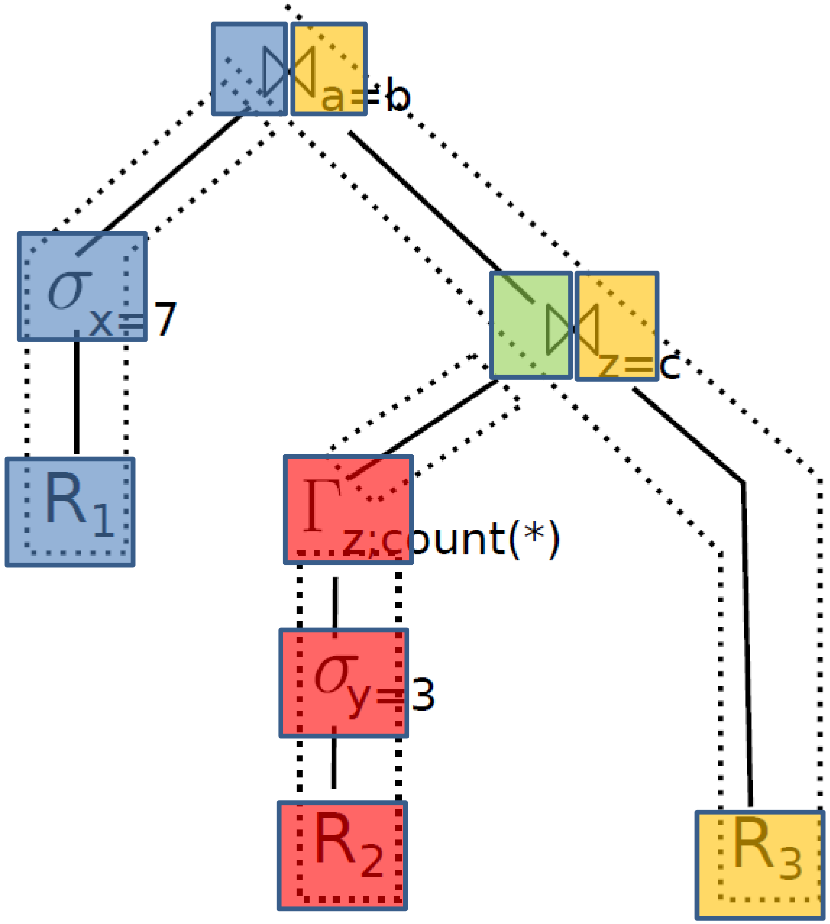
Push-based model for query compilation



- Data pushed up the pipeline
- Materializing only at *pipeline breakers*
- No function calls in loops => Compiler distributes data to registers and increases cache reuse.

Execute without "spilling data to memory"

Generate code for plan



for each tuple t in R_1
  if t.x = 7
    materialize t in hash table of $\bowtie_{a=b}$

for each tuple t in R_2
  if t.y = 3
    aggregate t in hash table of Γ_z

for each tuple t in Γ_z
  materialize t in hash table of $\bowtie_{z=c}$

for each tuple t_3 in R_3
  for each match t_2 in $\bowtie_{z=c}[t_3.\mathrm{c}]$
    for each match t_1 in $\bowtie_{a=b}[t_3.\mathrm{b}]$
      output t_1 o t_2 o t_3

Operator boundaries blurred – Imperative execution

The Catch

select d_tax from warehouse, district where w_id=d_w_id and w_zip='…'

```
define void @planStart(%14* %executionState) {
body:
  %0 = getelementptr inbounds %14* %executionState, i64 0, i32 0, i32 1,
       i64 0
  store i64 0, i64* %0, align 8
  %1 = getelementptr inbounds %14* %executionState, i64 0, i32 1
  call void @_ZN5hyper9HashTable5resetEv(%"hyper::HashTable"* %1)
  %2 = bitcast %14* %executionState to %"hyper::Database"**
  %3 = load %"hyper::Database"** %2, align 8
  %4 = getelementptr inbounds %"hyper::Database"* %3, i64 0, i32 1
  %5 = load i8** %4, align 8
  %warehouse = getelementptr inbounds i8* %5, i64 5712
  %6 = getelementptr inbounds i8* %5, i64 5784
  %7 = bitcast i8* %6 to i32**
  %8 = load i32** %7, align 8
  %9 = getelementptr inbounds i8* %5, i64 5832
  %10 = bitcast i8* %9 to %3**
  %11 = load %3** %10, align 8
  %12 = bitcast i8* %warehouse to i64*
  %size = load i64* %12, align 8
  %13 = icmp eq i64 %size, 0
  br i1 %13, label %scanDone, label %scanBody

scanBody:
  %tid = phi i64 [ 0, %body ], [ %34, %cont2 ]
  %14 = getelementptr i32* %8, i64 %tid
  %w_id = load i32* %14, align 4
  %15 = getelementptr inbounds %3* %11, i64 %tid, i32 0
  %16 = load i8* %15, align 1
  %17 = icmp eq i8 %16, 9
  br i1 %17, label %then, label %cont2

then:
  %w_zip = getelementptr inbounds %3* %11, i64 %tid, i32 1, i64 0
  %27 = call i32 @memcmp(i8* %w_zip, i8* @"string 137411111", i64 9)
  %28 = icmp eq i32 %27, 0
  br i1 %28, label %then1, label %cont2

then1:
  %29 = zext i32 %w_id to i64
  %30 = call i64 @llvm.x86.sse42.crc64.64(i64 0, i64 %29)
  %31 = shl i64 %30, 32
  %32 = call i8* @_ZN5hyper9HashTable15storeInputTupleEmj(%"hyper::
       HashTable"* %1, i64 %31, i32 4)
  %33 = bitcast i8* %32 to i32*
  store i32 %w_id, i32* %33, align 1
  br label %cont2

cont2:
  %34 = add i64 %tid, 1
  %35 = icmp eq i64 %34, %size
  br i1 %35, label %cont2.scanDone_crit_edge, label %scanBody

cont2.scanDone_crit_edge:
  %.pre = load %"hyper::Database"** %2, align 8
  %.phi.trans.insert = getelementptr inbounds %"hyper::Database"* %.pre,
       i64 0, i32 1
  %.pre11 = load i8** %.phi.trans.insert, align 8
  br label %scanDone

scanDone:
  %18 = phi i8* [ %.pre11, %cont2.scanDone_crit_edge ], [ %5, %body ]
  %district = getelementptr inbounds i8* %18, i64 1512
  %19 = getelementptr inbounds i8* %18, i64 1592
  %20 = bitcast i8* %19 to i32**
  %21 = load i32** %20, align 8
  %22 = getelementptr inbounds i8* %18, i64 1648
  %23 = bitcast i8* %22 to i64**
  %24 = load i64** %23, align 8
  %25 = bitcast i8* %district to i64*
  %size8 = load i64* %25, align 8
  %26 = icmp eq i64 %size8, 0
  br i1 %26, label %scanDone6, label %scanBody5
```

(more)

`select d_tax from warehouse, district where w_id=d_w_id and w_zip='…'`

```
scanBody5:
  %tid9 = phi i64 [ 0, %scanDone ], [ %58, %loopDone ]
  %36 = getelementptr i32* %21, i64 %tid9
  %d_w_id = load i32* %36, align 4
  %37 = getelementptr i64* %24, i64 %tid9
  %d_tax = load i64* %37, align 8
  %38 = zext i32 %d_w_id to i64
  %39 = call i64 @llvm.x86.sse42.crc64.64(i64 0, i64 %38)
  %40 = shl i64 %39, 32
  %41 = getelementptr inbounds %14* %executionState, i64 0, i32 1, i32 0
  %42 = load %"hyper::HashTable::Entry"*** %41, align 8
  %43 = getelementptr inbounds %14* %executionState, i64 0, i32 1, i32 2
  %44 = load i64* %43, align 8
  %45 = lshr i64 %40, %44
  %46 = getelementptr %"hyper::HashTable::Entry"** %42, i64 %45
  %47 = load %"hyper::HashTable::Entry"** %46, align 8
  %48 = icmp eq %"hyper::HashTable::Entry"* %47, null
  br i1 %48, label %loopDone, label %loop

loopStep:
  %49 = getelementptr inbounds %"hyper::HashTable::Entry"* %iter, i64 0,
        i32 1
  %50 = load %"hyper::HashTable::Entry"** %49, align 8
  %51 = icmp eq %"hyper::HashTable::Entry"* %50, null
  br i1 %51, label %loopDone, label %loop

loop:
  %iter = phi %"hyper::HashTable::Entry"* [ %47, %scanBody5 ], [ %50, %
        loopStep ]
  %52 = getelementptr inbounds %"hyper::HashTable::Entry"* %iter, i64 1
  %53 = bitcast %"hyper::HashTable::Entry"* %52 to i32*
  %54 = load i32* %53, align 4
  %55 = icmp eq i32 %54, %d_w_id
  br i1 %55, label %then10, label %loopStep

then10:
  call void @_ZN6dbcore16RuntimeFunctions12printNumericEljj(i64 %d_tax,
        i32 4, i32 4)
  call void @_ZN6dbcore16RuntimeFunctions7printNlEv()
  br label %loopStep

loopDone:
  %58 = add i64 %tid9, 1
  %59 = icmp eq i64 %58, %size8
  br i1 %59, label %scanDone6, label %scanBody5

scanDone6:
  ret void
}
```

Low-level, error-prone coding

Query compilation

- Pipelined query processing without interpretation cost
- Very painful to implement
- **BUT:** Benefits have led major DBMS (and Spark!) to implement it

Conclusion

The ***processing model*** of a DBMS defines how the system executes a query plan.

- Tuple-at-a-time via the **iterator model**
- Query compilation
- Vectorization model
- **Block-oriented model** (typically column-at-a-time)

Hybrids do exist!!!