CS460

Systems for Data Management and Data Science

Query Execution

Prof. Anastasia Ailamaki

Data-Intensive Applications and Systems (DIAS) Lab

"It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures"
– Alan Perlis

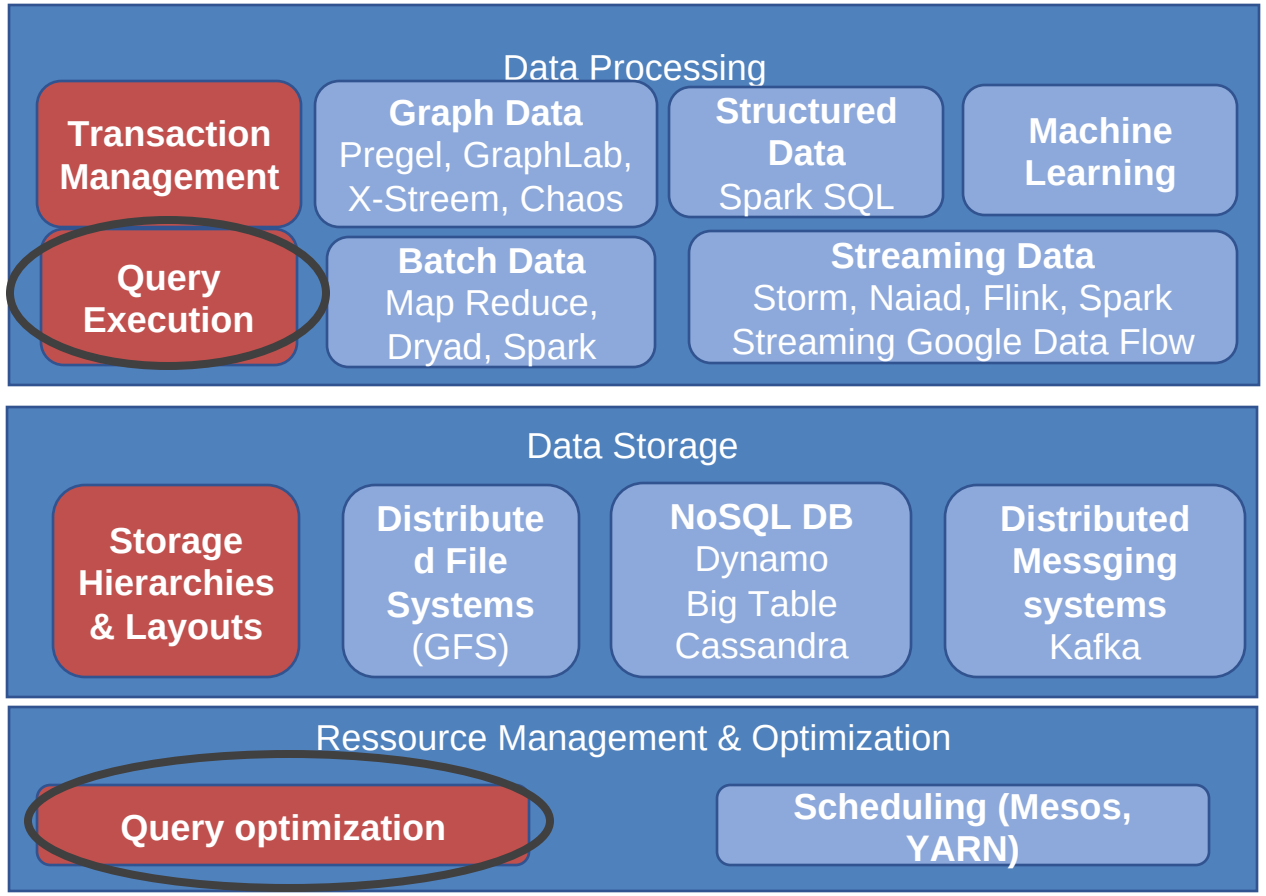
Today's topic

Gossip Protocols

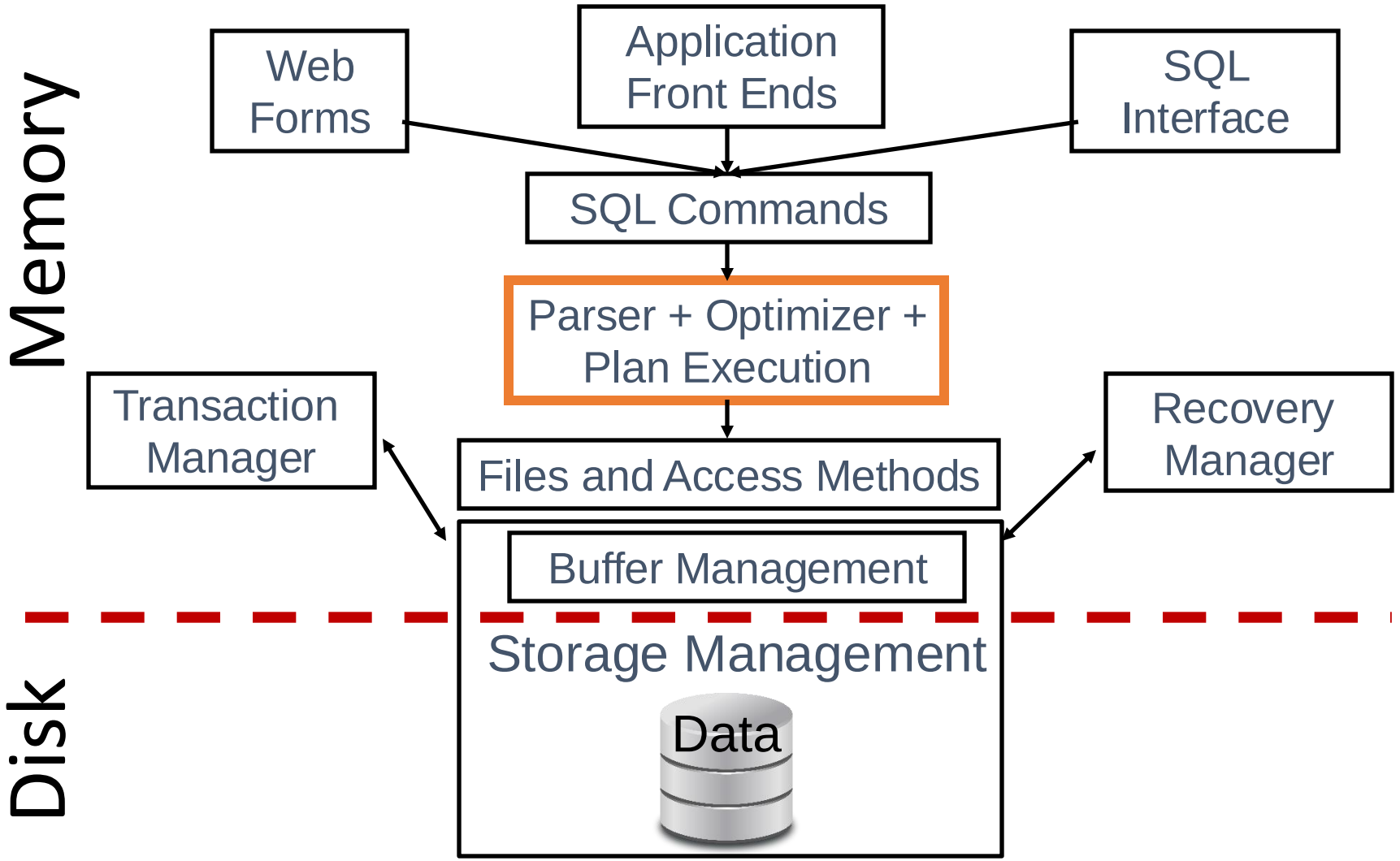
Consistency protocols
CAP Theorem

Distributed/Decentralized systems

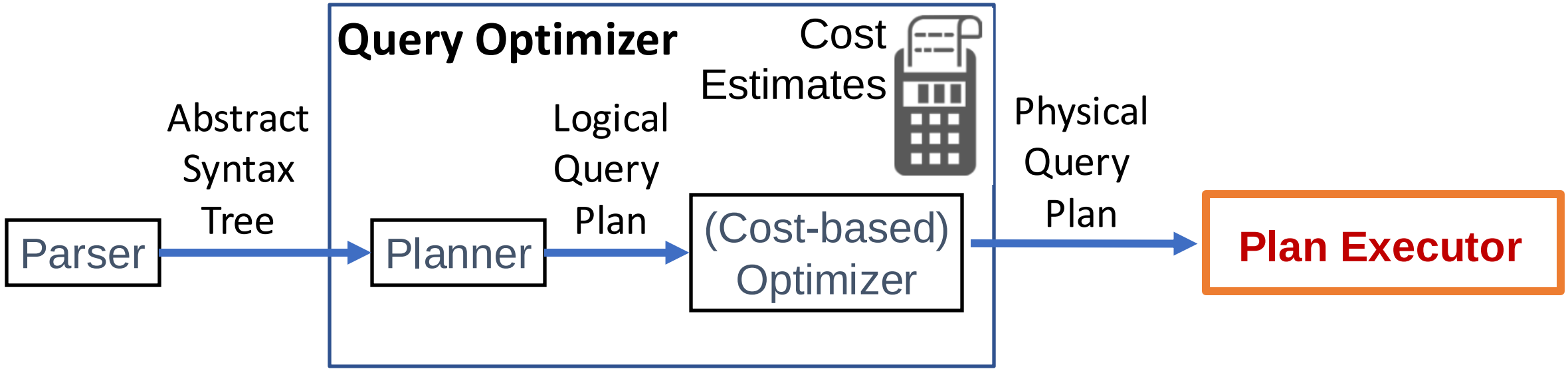
Data science software stack



(Simplified) DBMS Architecture



Zoom in: Query Planner/Optimizer/Executor



How the DBMS executes a query (plan)

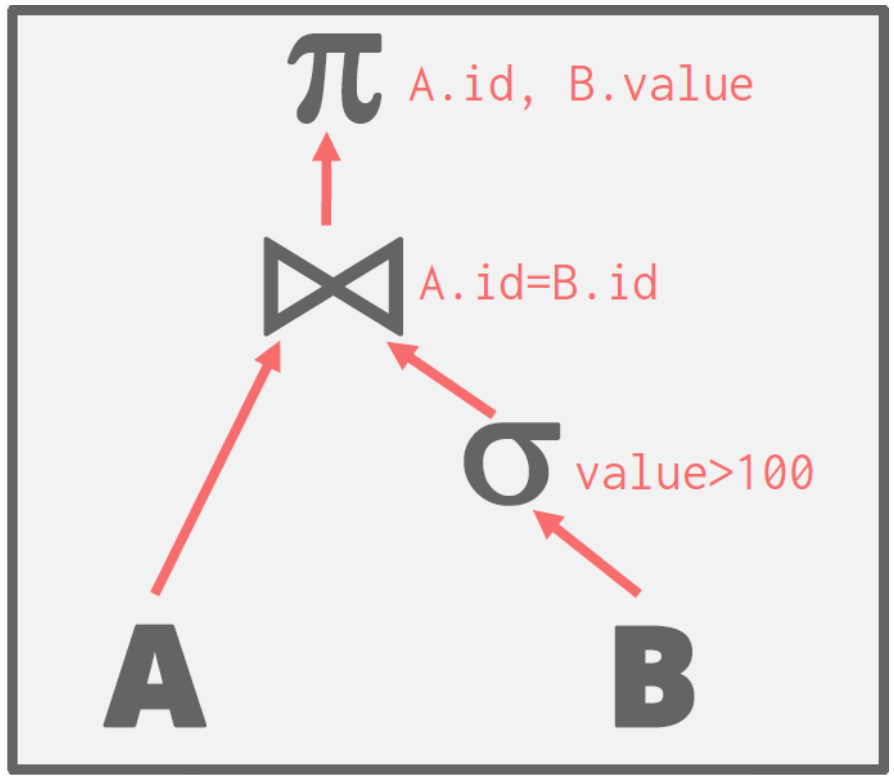
Query Plan

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
```



Composable algebra => composable execution

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 (Volcano Model)

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

```
class Operator:
  Optional<Tuple> next()
```

- On each invocation, the operator returns either a single tuple or a marker that there are no more tuples
- **next** calls **next** on the operator's children to retrieve and process their tuples

```
class Project:
  Operator input, Expression proj
  Optional<Tuple> next():
    t = input.next()
    if (t empty) return empty
    return proj(t)
```

```
class Filter:
  Operator input, Expression pred

  Optional<Tuple> next():
    while (true):
      t = input.next()
      if (t empty or pred(t)) return t
```

Common operator interface => composability

Notation

```
class Operator:
  Optional<Tuple> next()
```

```
class Operator:
  generator<Tuple> Next()
```

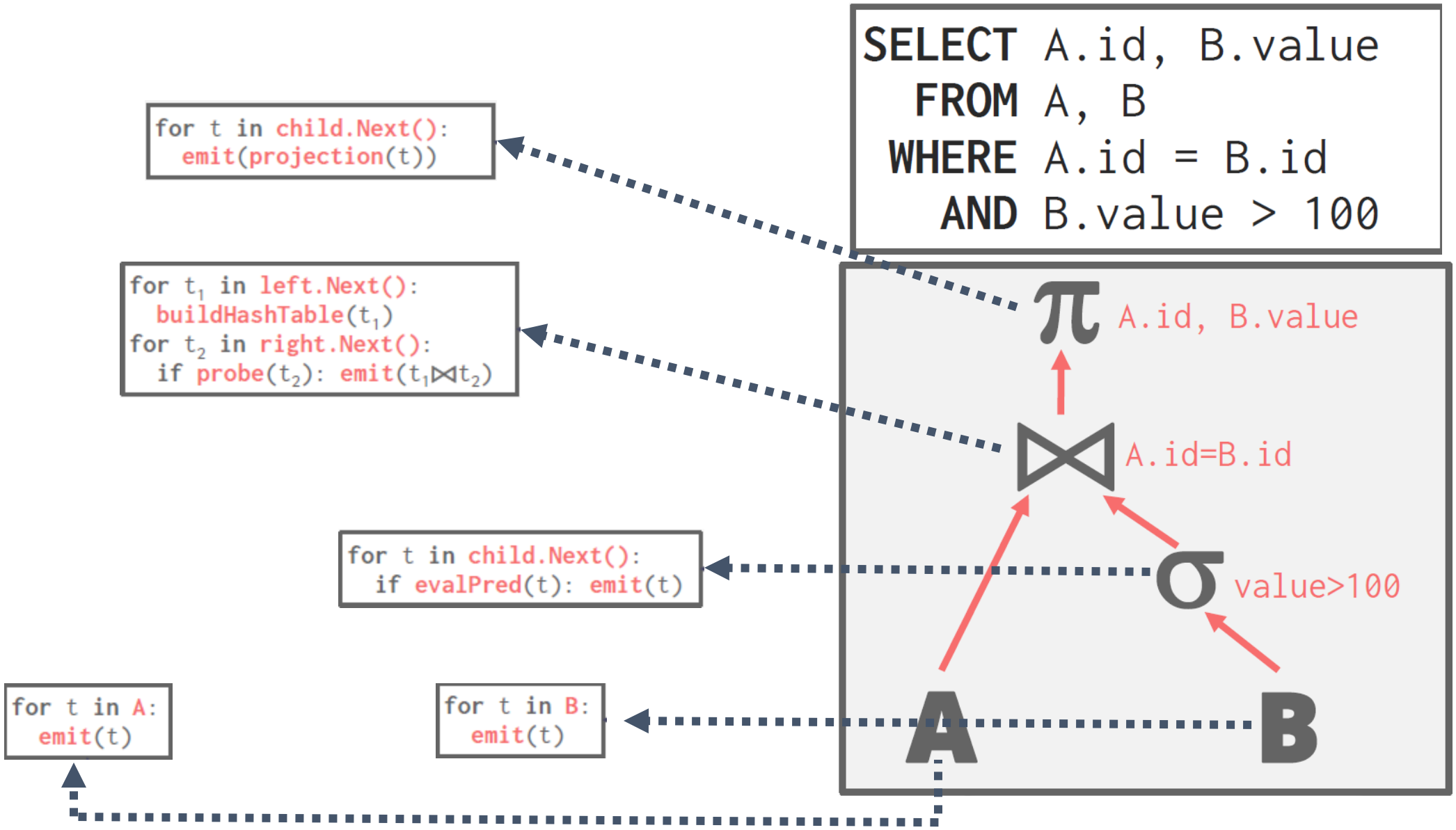
```
class Filter:
  Operator input, Expression pred

  Optional<Tuple> next():
    while (true):
      t = input.next()
      if (t empty or pred(t)) return t
```

```
class Filter:
  Operator input, Expression pred

  generator<Tuple> next():
    for t in input.Next():
      if pred(t) emit t
```

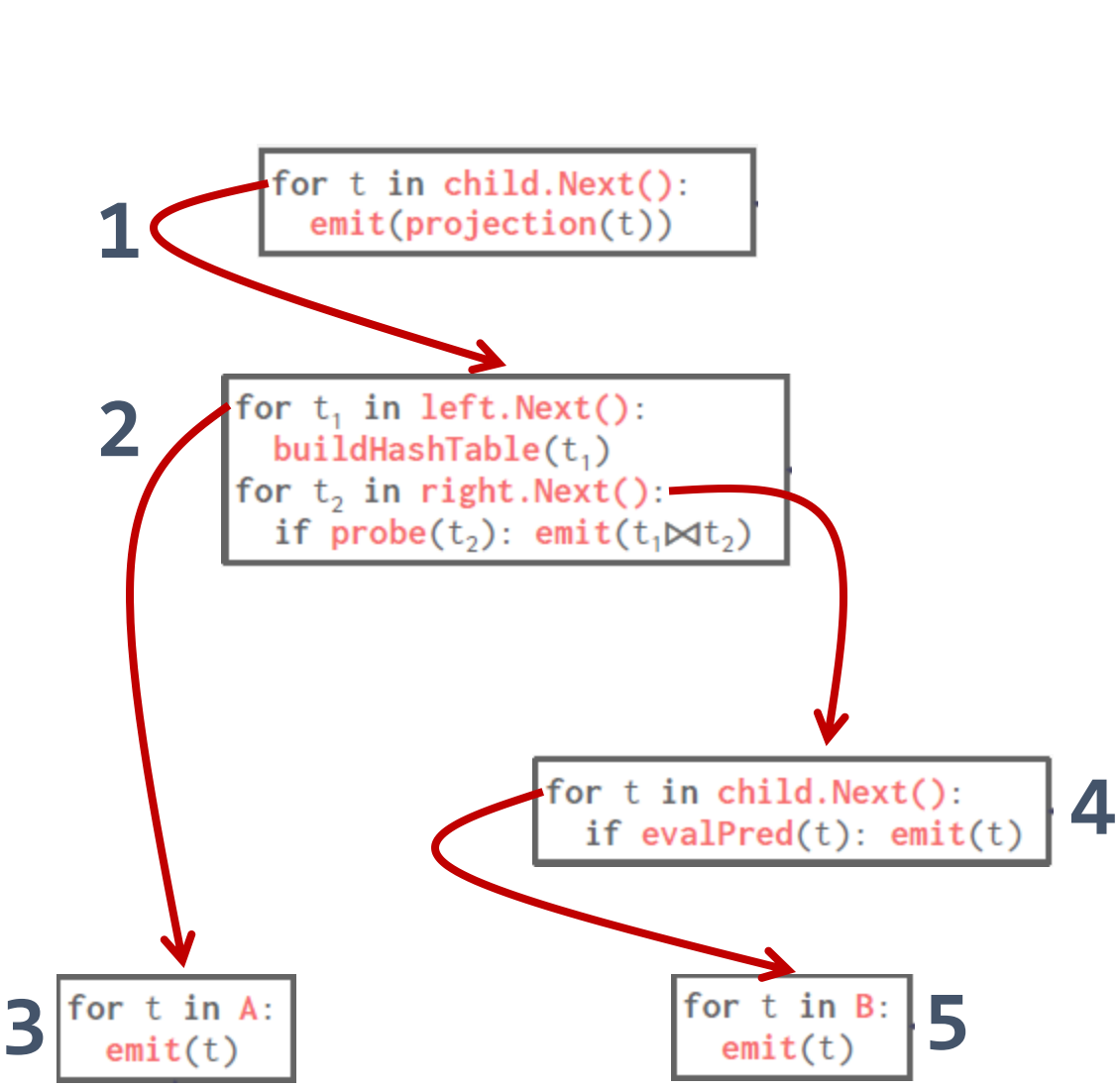
```
class Project:
  Operator input, Expression proj
  Optional<Tuple> next():
    t = input.next()
    if (t empty) return empty
    return proj(t)
```

```
class Project:
  Operator input, Expression proj

  generator<Tuple> Next():
    for t in input.Next():
      emit proj(t)
```

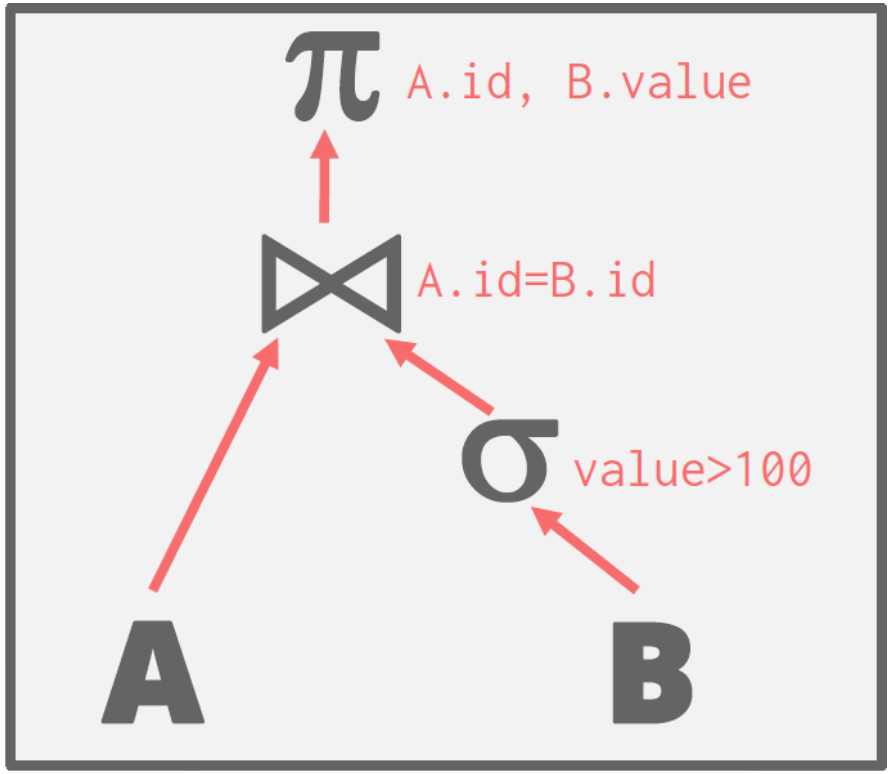
Example: Iterator Model



Example: Iterator Model (cont)




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



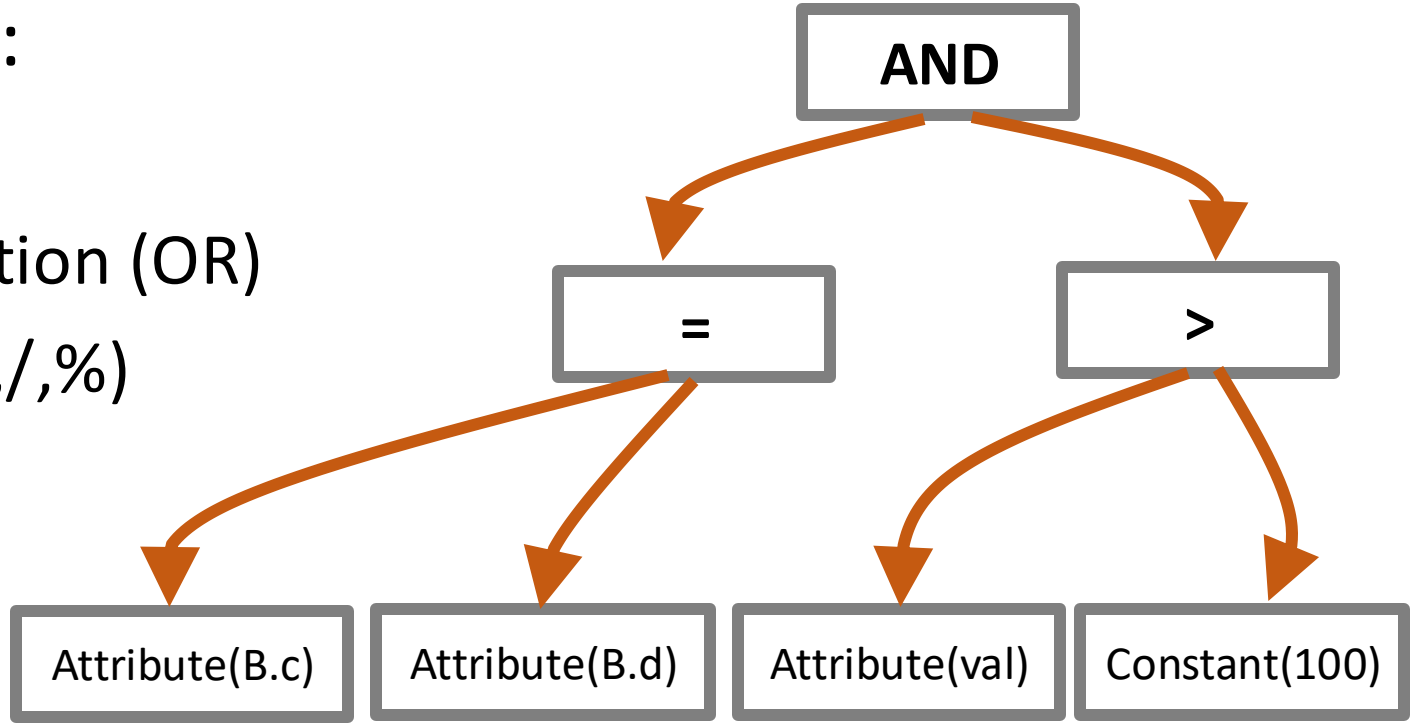
(Interpreted) Expression Evaluation

```
class Filter:
  Operator input
  Expression pred = (B.c = B.d) AND (B.value > 100)

  generator<Tuple> next(): …
```

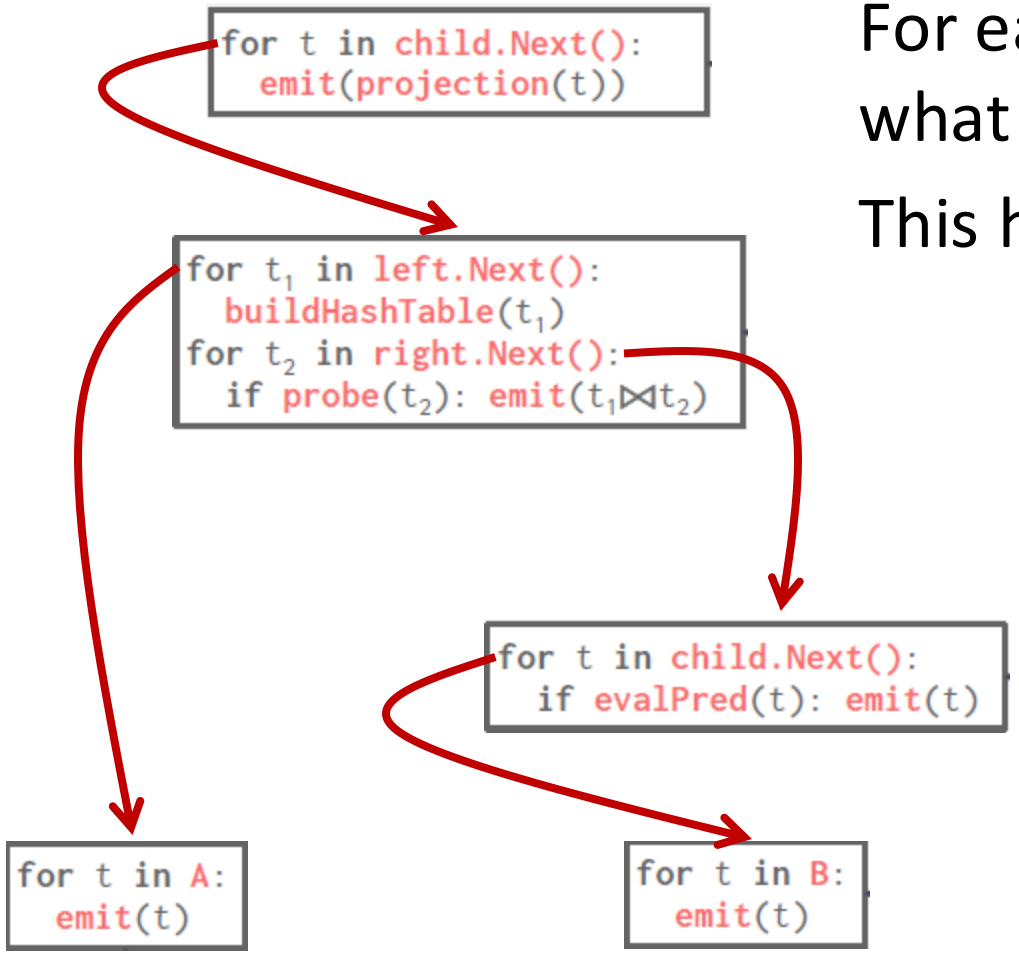
```
SELECT A.id, B.value
  FROM A, B
WHERE A.id = B.id
    AND B.c = B.d
    AND B.value > 100
```

Nodes in the tree represent different expression types:

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



Interpreted, tuple-at-a-time processing



The DBMS traverses the tree.
For each node that it visits, it has to figure out what the operator needs to do. Same for expressions.

This happens for every… single… tuple…

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

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.
- Often bottom-up plan processing.

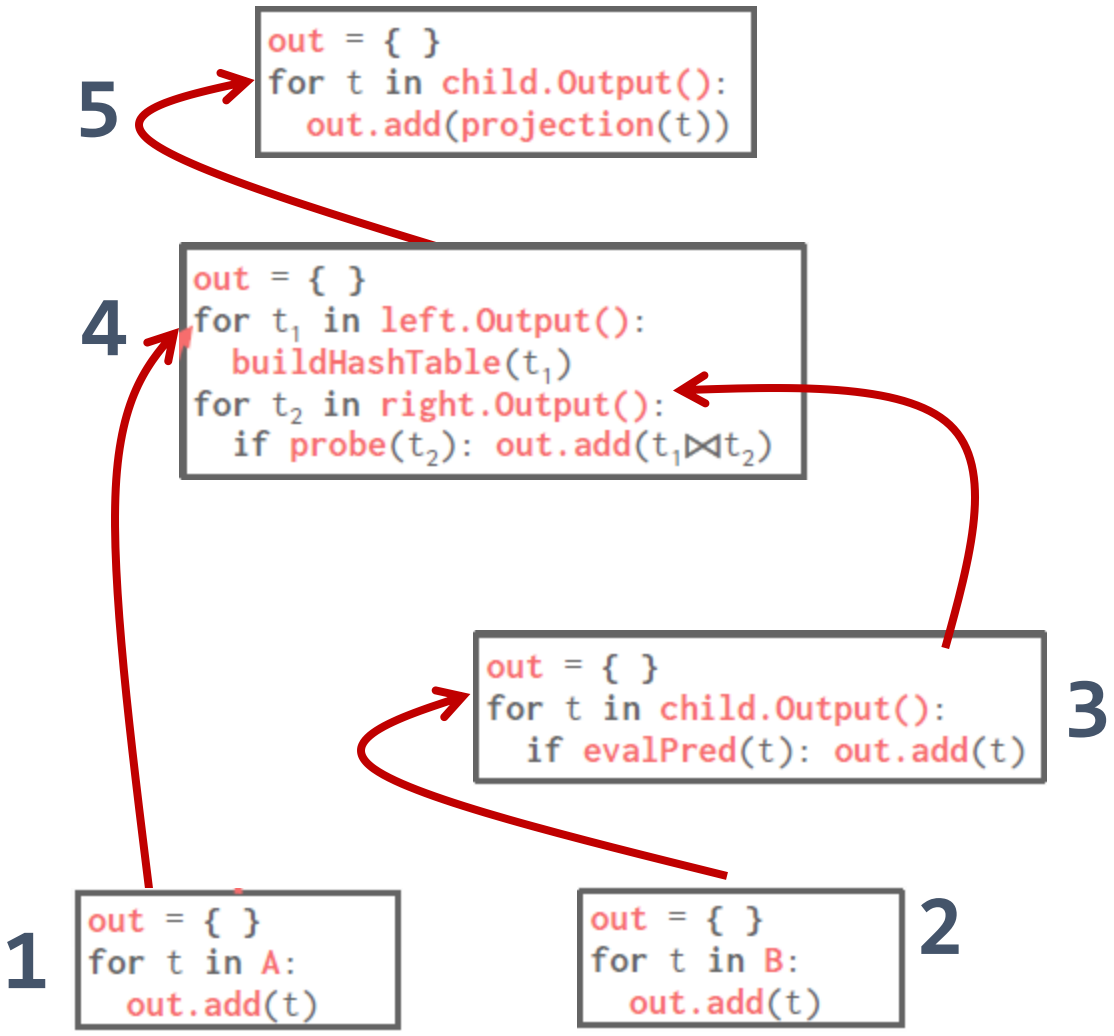
```
class Operator:
  Tuples Output()
```

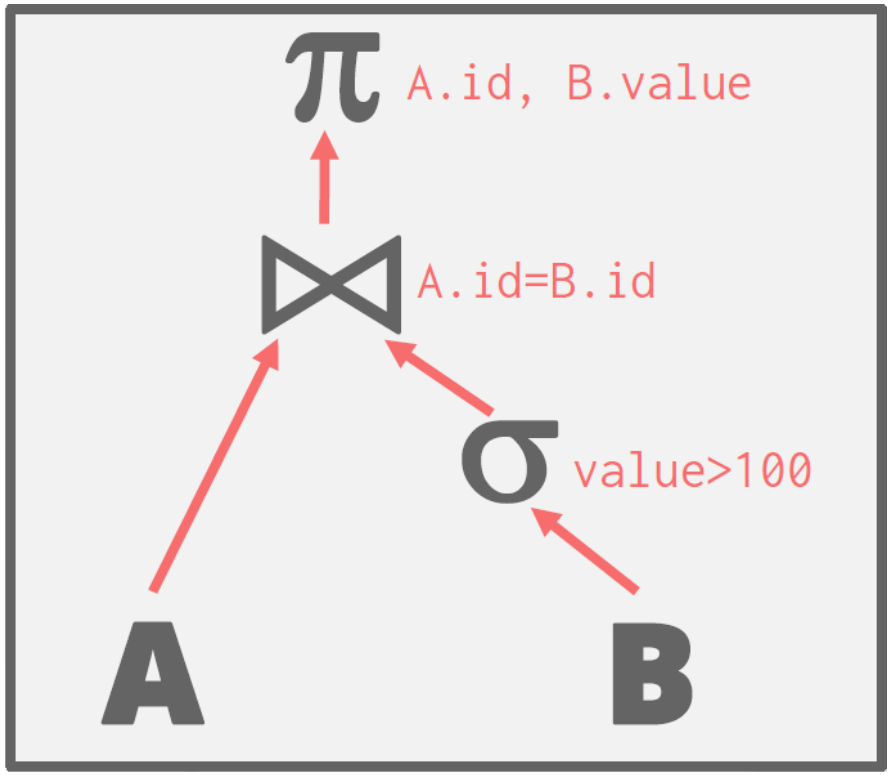
```
class Project:
  Operator input, Expression proj
  Tuples Output():
    out = {}
    for t in input.Output():
      out.append(proj(t))
    return out
```

```
class Filter:
  Operator input, Expression pred

  Tuples Output():
    out = {}
    for t in input.Output():
      if pred(t) out.append(t)
    return out
```

Block-oriented Model

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





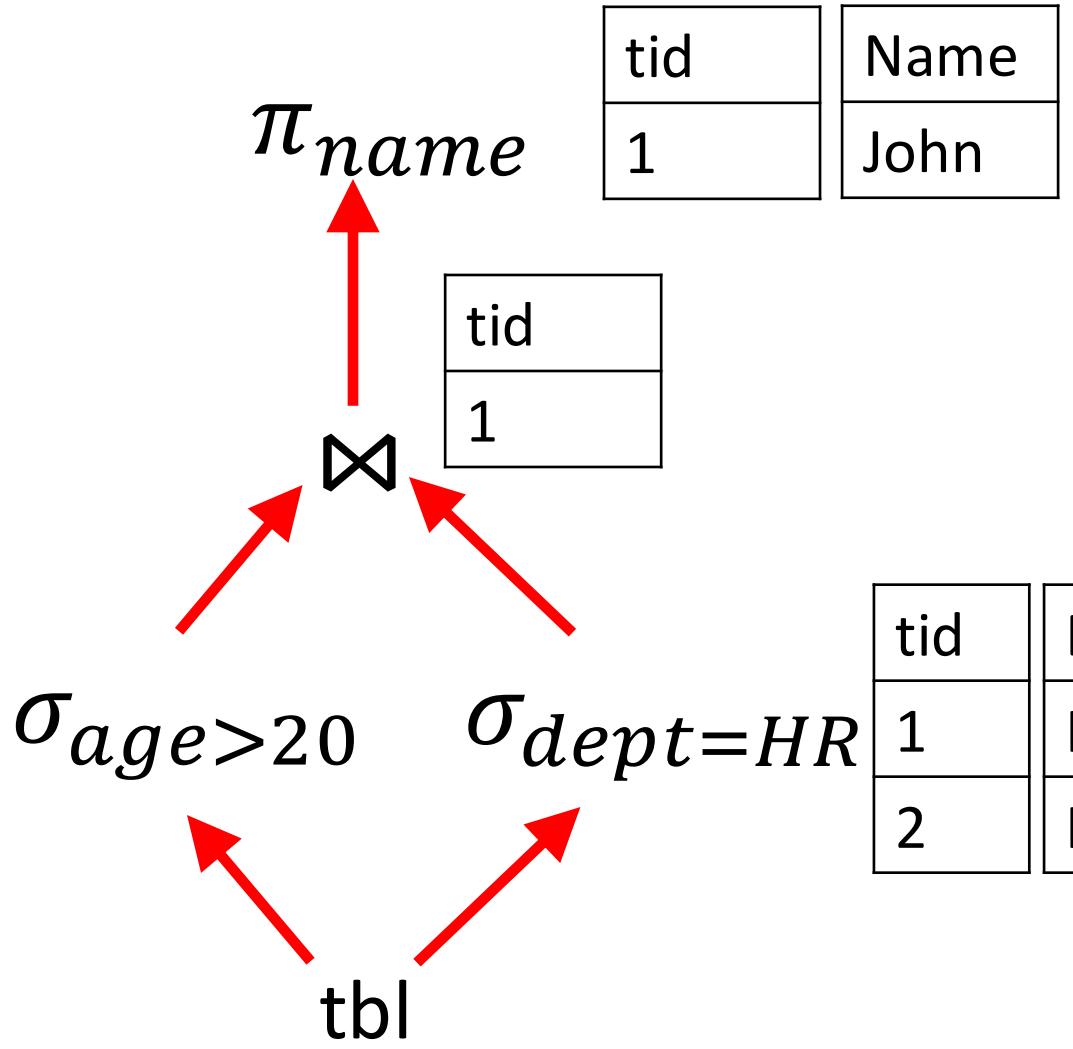
The (output) materialization problem – Naïve version

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”
```

tid	Age
1	22
3	37



tid	Name
1	John

tid
1

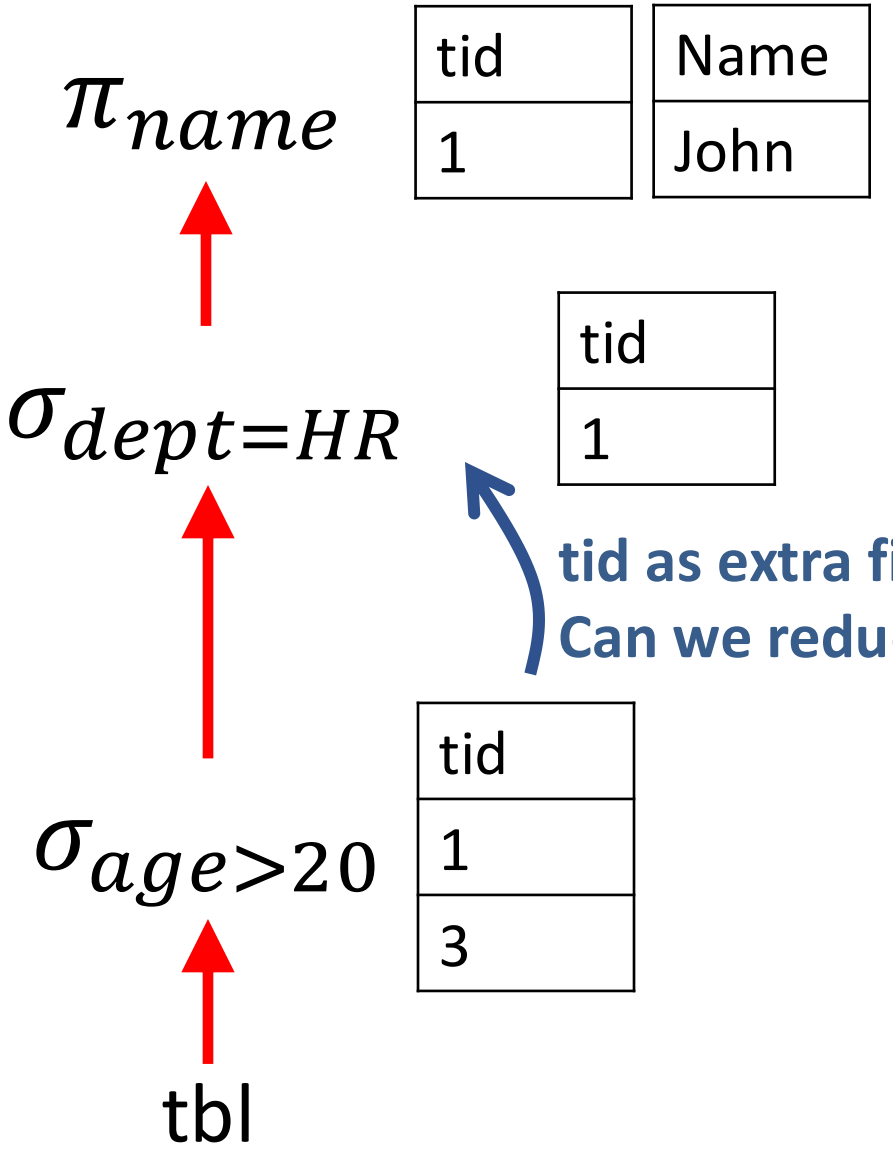
tid	Dept
1	HR
2	HR

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"
```

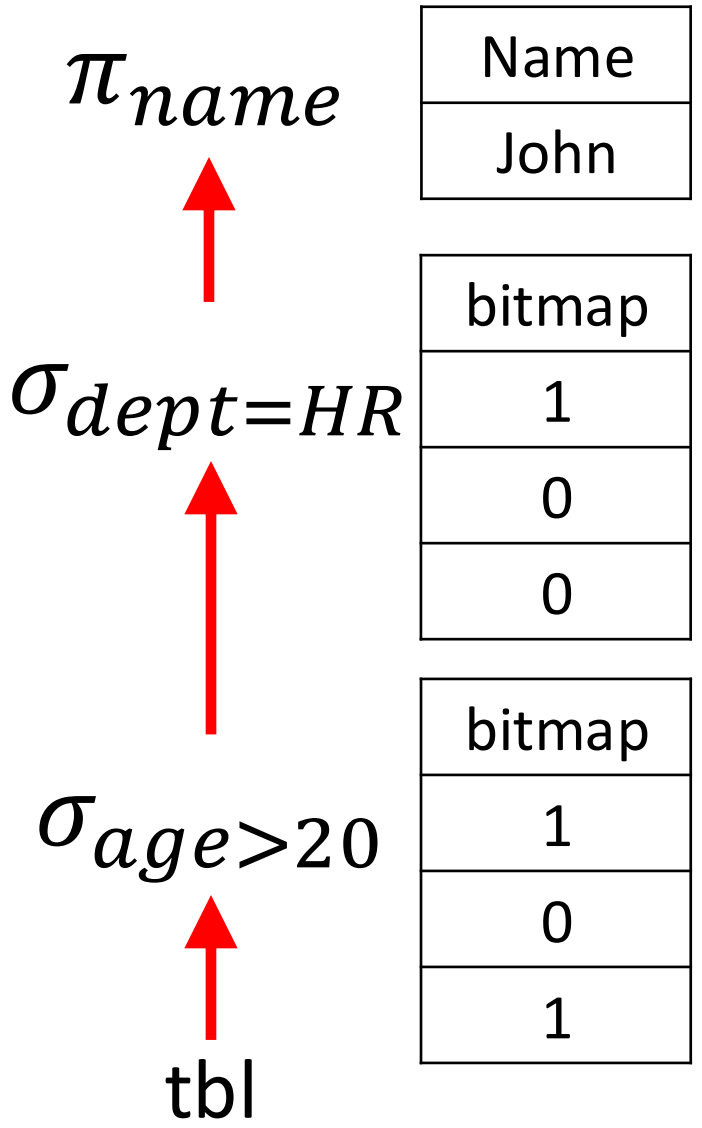


The (output) materialization problem – selection vector

```
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



Name
John

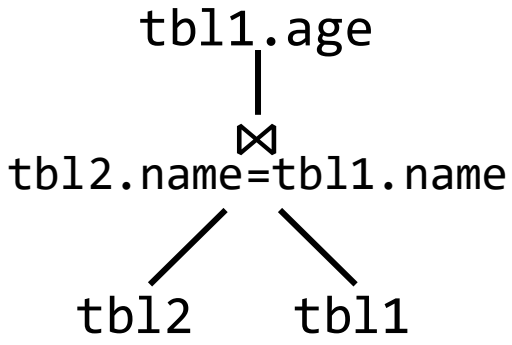
bitmap
1
0
0

bitmap
1
0
1

- **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

Solution 1: Stitch columns before join

Solution 2: Sort list of tids before projection

Solution 3: Use order-preserving join algorithm (eg jive-join) – but 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)

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 in fridge

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

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**
- Middle Ground: Vectorization model
- Extreme II: **Block-oriented model** (typically column-at-a-time)

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

The middle ground: Vectorization model

- Like iterator model, each operator implements a **next** function
- Each operator emits a **vector** of tuples instead of a single tuple

```
class Operator:
  Optional<Vector<Tuple>> next()
```

```
class Project:
  Operator input, Expression proj
  Optional<Vector<Tuple>> next():
    vec = input.next()
    if (vec empty) return empty
    out = {}
    for t in vec:
      out.add(proj(t))
    return out
```

```
class Filter:
  Operator input, Expression pred

  Optional<Vector<Tuple>> next():
    while (true):
      vec = input.next()
      if (vec empty) return vec
      out = {}
      for t in vec:
        if pred(t): out.add(t)
      return out
```

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
- Basic model commercialized by 

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 instructions

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)

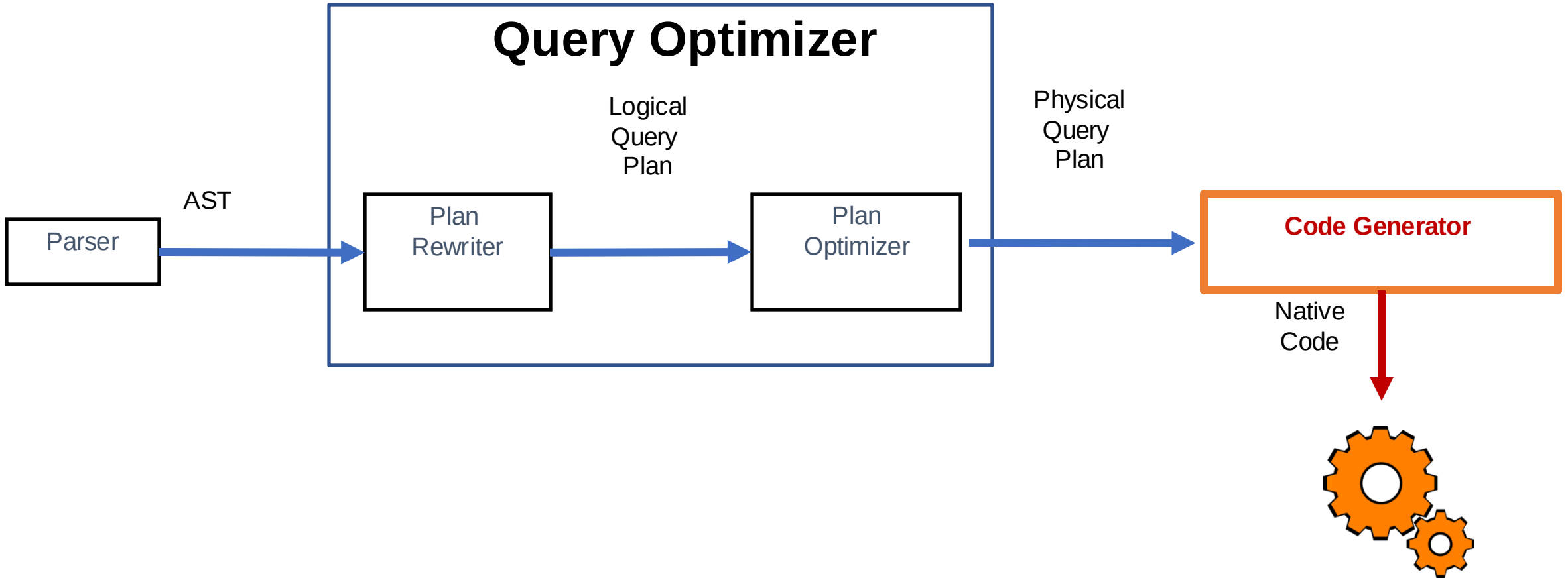
Move from general to specialized code

- CPU-intensive code parts 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!
  - Attribute types

    => (Inline) pointer casting instead of data access (virtual) function calls
  - Query predicate types

    => data comparisons

Query Compiler



Two approaches for code generation

Transpilation

- DBMS converts a query plan into imperative source code
- Compile the produced code to generate native code with a conventional compiler

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 plan.
  → Bake in all the predicates and type conversions.
- Advantages:
  - Fewer function calls during query evaluation
  - Generated code uses cache-resident data more efficiently
  - Compiler optimization techniques come free
- Off-the-shelf compiler converts code into a shared object, links it to the DBMS process, and then invokes the exec function.

Operator templates

```
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 – pull values up
2. For tuple values, calculate the offset of the target attribute
3. Resolve datatype (switch / virtual call)

Templated plan

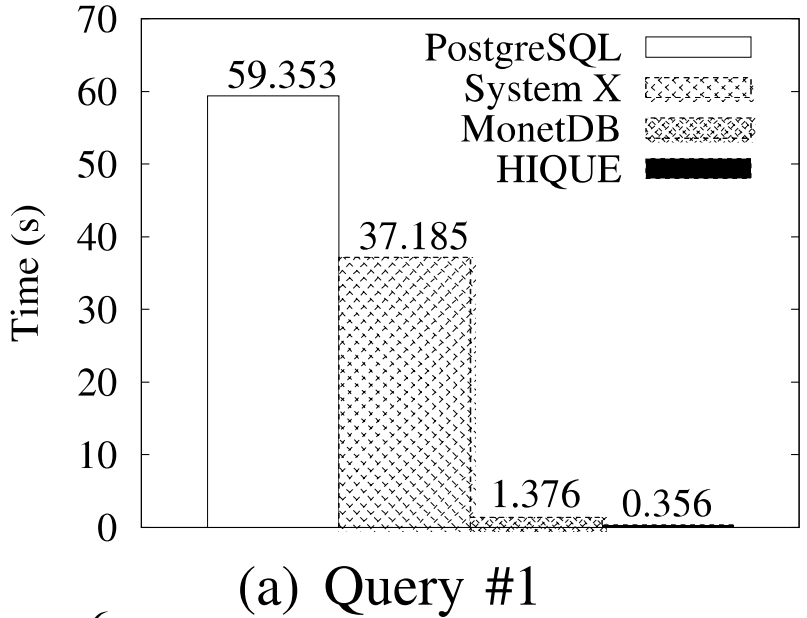
Known at query compile time

```
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)
```

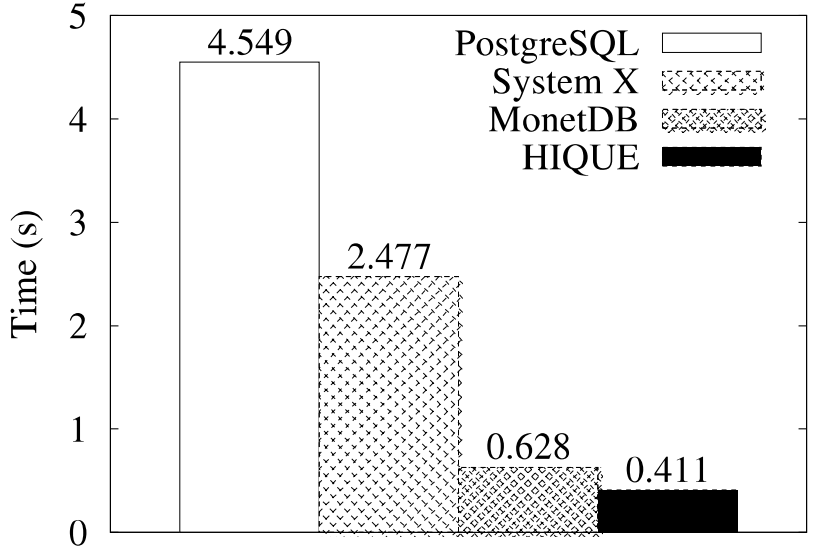
Integrating with the rest of the DBMS

- 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

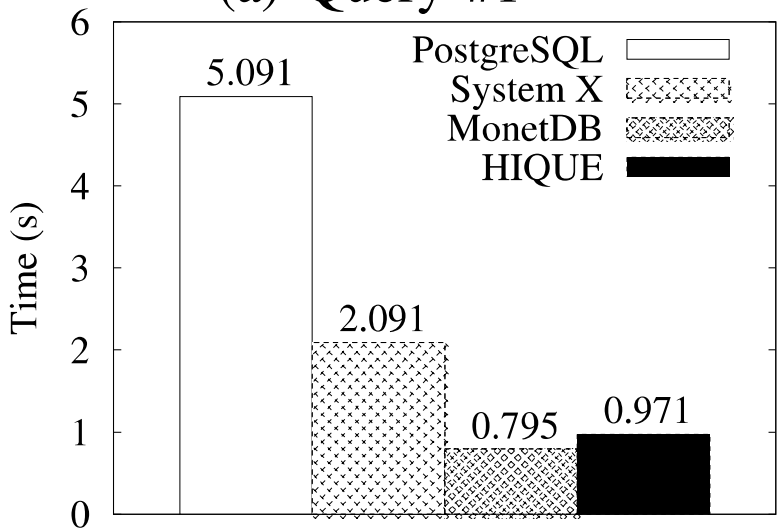
Indicative Performance [Krikellas, ICDE 2010]



(a) Query #1



(b) Query #3



(c) Query #10

Up to 2 orders of magnitude reported improvement when compared to interpreted DBs (e.g., PostgreSQL)

The catch [Krikellas, ICDE 2010]

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 type-checking, smaller code, promote cache reuse

BUT

- Compilation takes time
- Sticks to the operator “legacy” abstraction

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 plan.
  → Bake in all the predicates and type conversions.
- Advantages:
  - Fewer function calls during query evaluation
  - Generated code uses cache-resident data more efficiently
  - Compiler optimization techniques come free
- Off-the-shelf compiler converts code into a shared object, links it to the DBMS process, and then invokes the exec function.

Operator templates

```
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 – pull values up
2. For tuple values, calculate the offset of the target attribute
3. Resolve datatype (switch / virtual call)

Templated plan

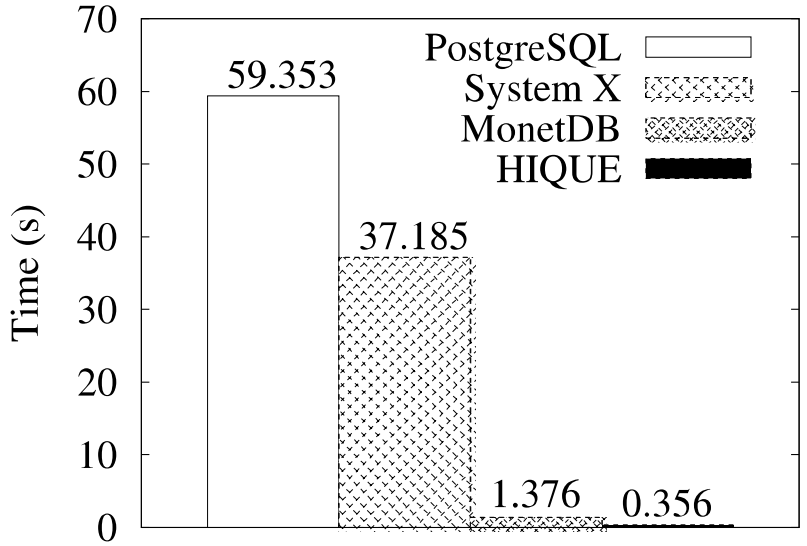
Known at query compile time

```
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)
```

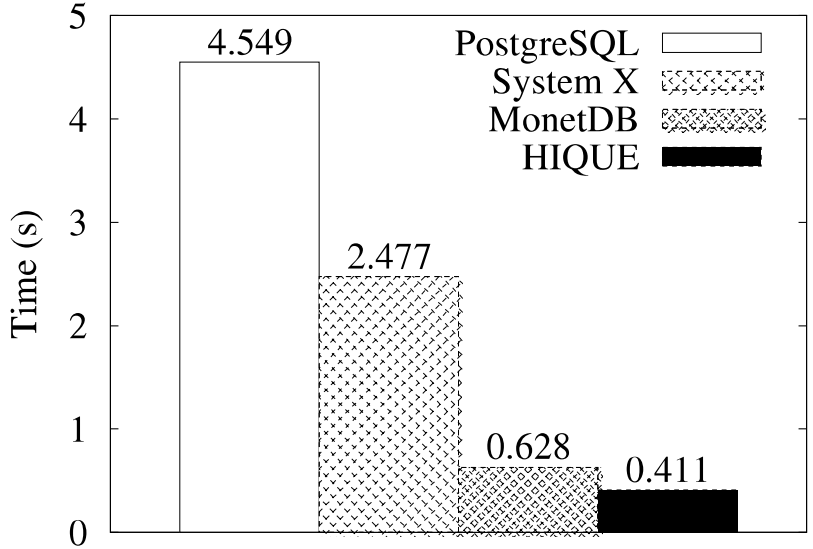
Integrating with the rest of the DBMS

- 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

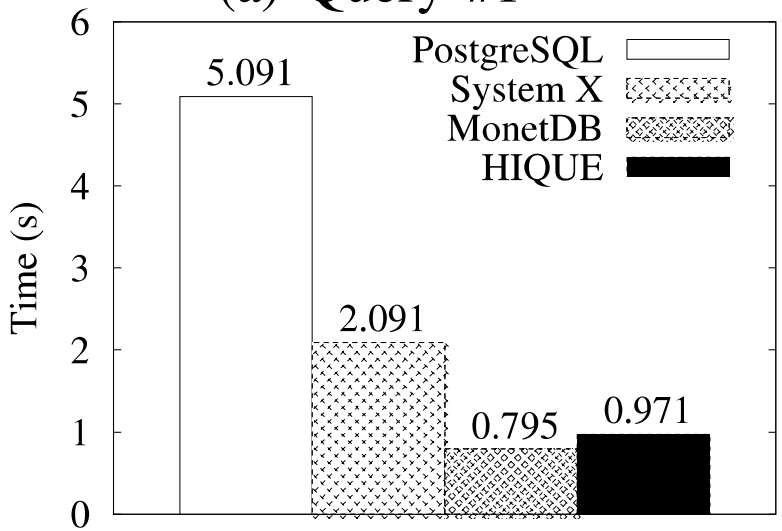
Indicative Performance [Krikellas, ICDE 2010]



(a) Query #1



(b) Query #3



(c) Query #10

Up to 2 orders of magnitude reported improvement when compared to interpreted DBs (e.g., PostgreSQL)

The catch [Krikellas, ICDE 2010]

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 type-checking, smaller code, promote cache reuse

BUT

- Compilation takes time
- Sticks to the operator “legacy” abstraction

Two approaches for code generation

Transpilation

- DBMS converts a query plan into imperative source code
- Compile the produced code to generate native code with a conventional compiler

JIT compilation

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

Reminder: Operator templates

```
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)
```

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)
```

Operator boundaries

```
SELECT A.a + B.b
FROM A, B
WHERE A.val = ? + 1
   AND B.c = A.d
```

```
Main:
   execute(Op-1)
   execute(Op-2)
   …
   execute(Op-n)
```

```
tuple_size = ###
predicate_offset = ###
(val_offset)

for t in
range(table.num_tuples):
  tuple = table.data + t *
tuple_size
  val = *(tuple+predicate_offset)
```

σ

```
tuple2_size = ###
key_offset = ### (d_offset)
for t in
range(emitted.num_tuples):
  t2 = emitted.data + t *
tuple2_size
  k = hash(*(t2+key_offset) )
```

⋈

Generated code: more specialization

```
SELECT A.a + B.b
FROM A, B
WHERE A.val = ? + 1
   AND B.c = A.d
```

```
tuple_size = ###
predicate_offset = ### (val_offset)
parameter_value = ###
key_offset = ### (d_offset)

for t in range(table.num_tuples):
  tuple = table.data + t * tuple_size
  val = *(tuple+predicate_offset)
  if (val == parameter_value + 1):
k = hash(*(tuple+key_offset)
while (probe_ht(k)):
   emit(t2, probe_match)
```

```
tuple_size = ###
predicate_offset = ###
(val_offset)
parameter_value = ###

for t in
range(table.num_tuples):
  tuple = table.data + t *
tuple_size
```

```
tuple2_size = ###
key_offset = ### (d_offset)
for t in
range(emitted.num_tuples):
  t2 = emitted.data + t *
tuple2_size
  k = hash(*(t2+key_offset) )
```

Operator abstractions – The good and the bad

- **Composability** – Express complex logic using small modules
- **Modularity** – Develop each component independently
- **Artificial boundaries** – Cost of modularity (?)

Operator abstractions – Through the looking glass

Functionality & DBMS Dev

Composability

Express complex logic using small modules

Modularity

Develop each component independently

Query Execution

Artificial boundaries

Operator Fusion

Functionality & DBMS Dev

Composability

Express complex logic using small modules

Modularity

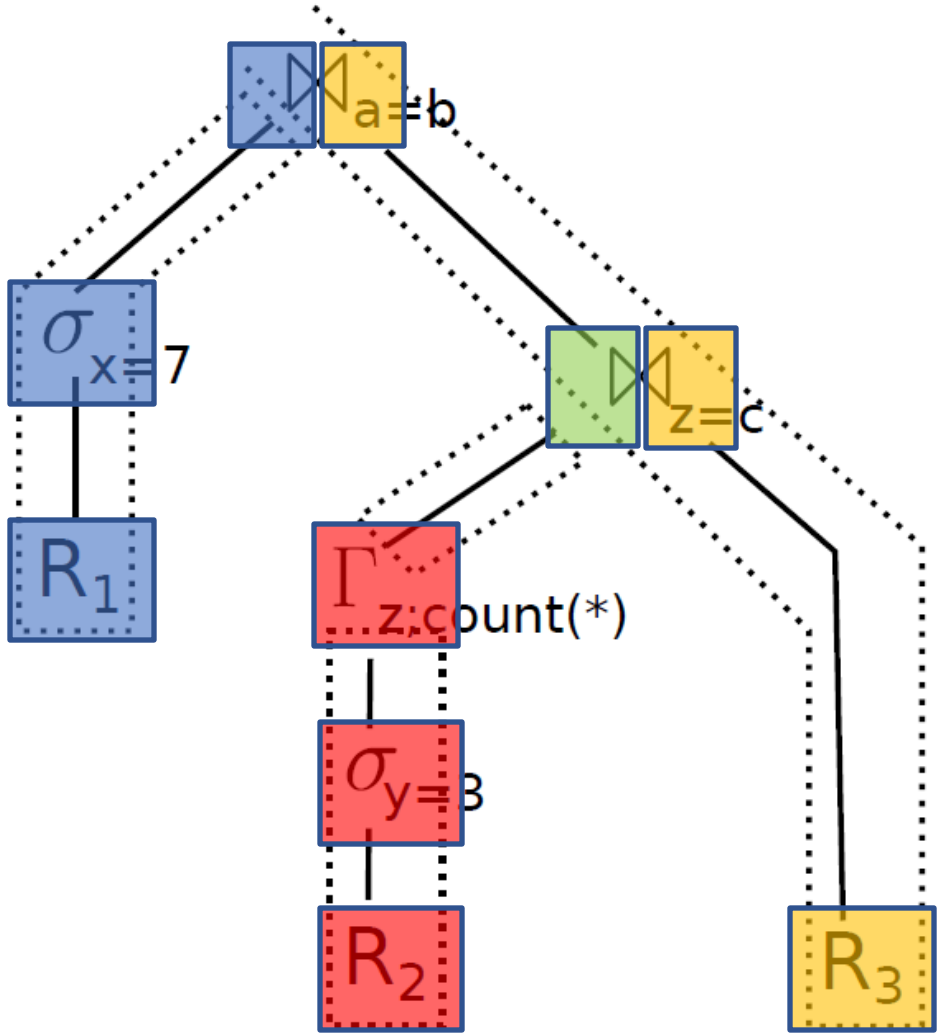
Develop each component independently

Query Execution

Blurred operator boundaries

HyPer

Generate code for plan [HyPer]



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 and materialize into 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

Push-based model for query compilation

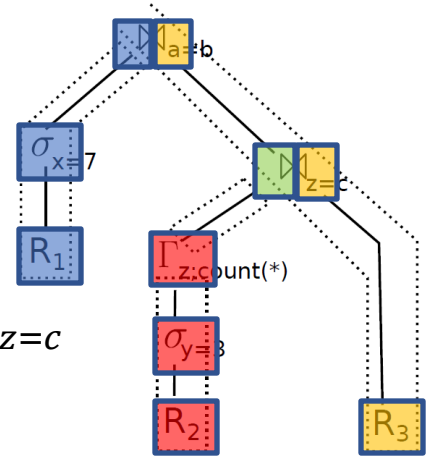
- Data pushed up the pipeline
- Materializing only at *pipeline breakers*
- No function calls in loops => Compiler distributes data to registers and increases cache reuse.

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

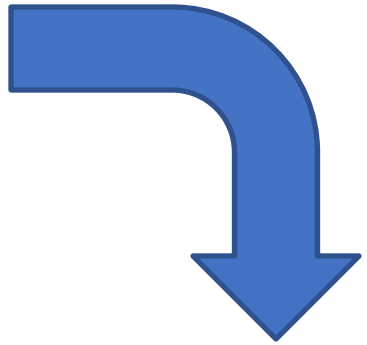


JIT compilation: The HyPer approach

- HyPer goal: “Keep a tuple in CPU registers as long as possible”
  - Push data through execution plan
  - **Blur operator boundaries**
- 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.

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
}
```

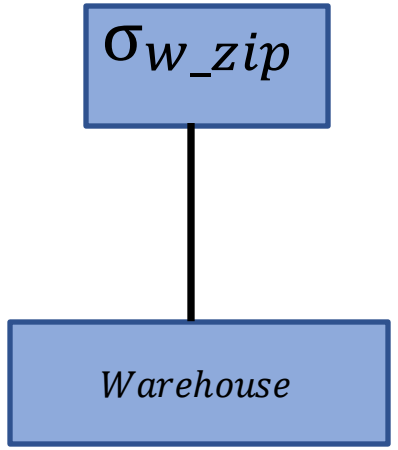
- Produced code very close to assembly
- Compilation very fast (tens of milliseconds!)

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
```



(more)

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

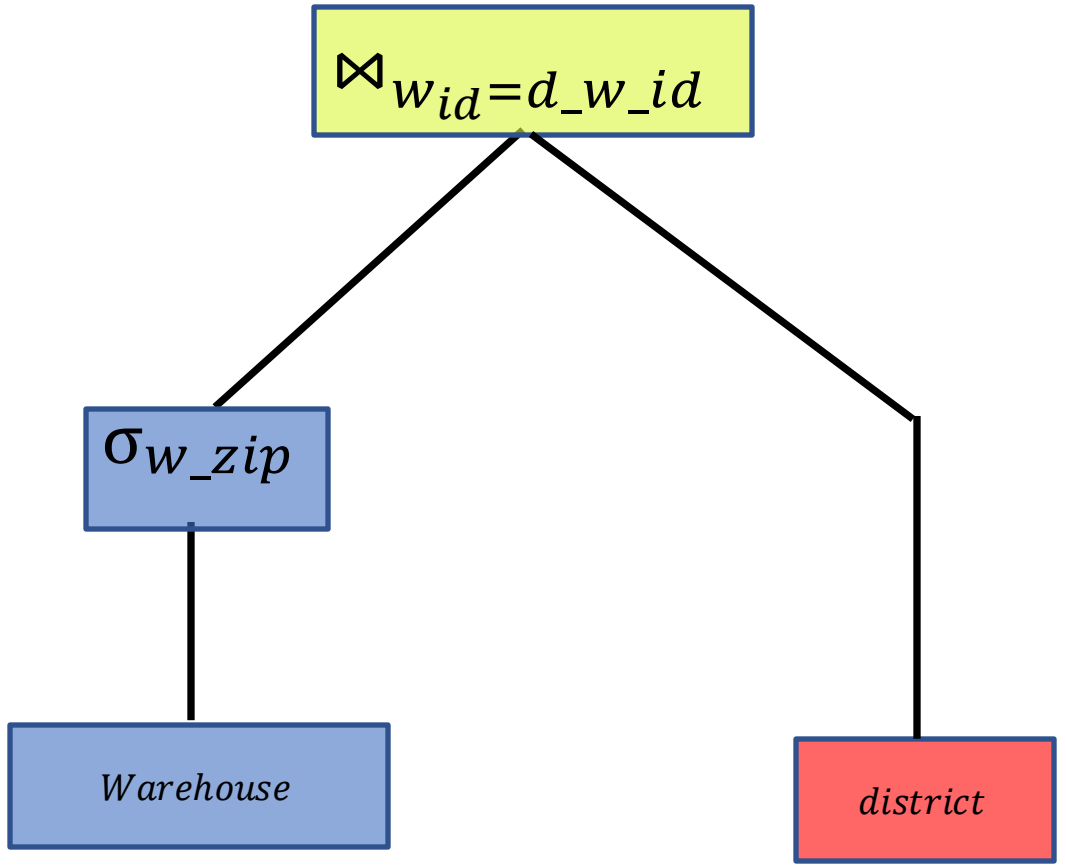
```
  %30 = call i64 @llvm.x86.sse42.crc64.64(i64 0, i64 %29)
  %31 = shl i64 %30, 32
  %32 = call i8* @_ZN5hyper9HashTable15storeInputTupleEmj(%"hyper::
      HashTable"* %11, i64 %31, i32 4)
  %33 = bitcast i8* %32 to i32*
  store i32 %w_id, i32* %33, align 1
  br label %cont2

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
```

```
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
```

(Even more!)

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

```
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
}
```

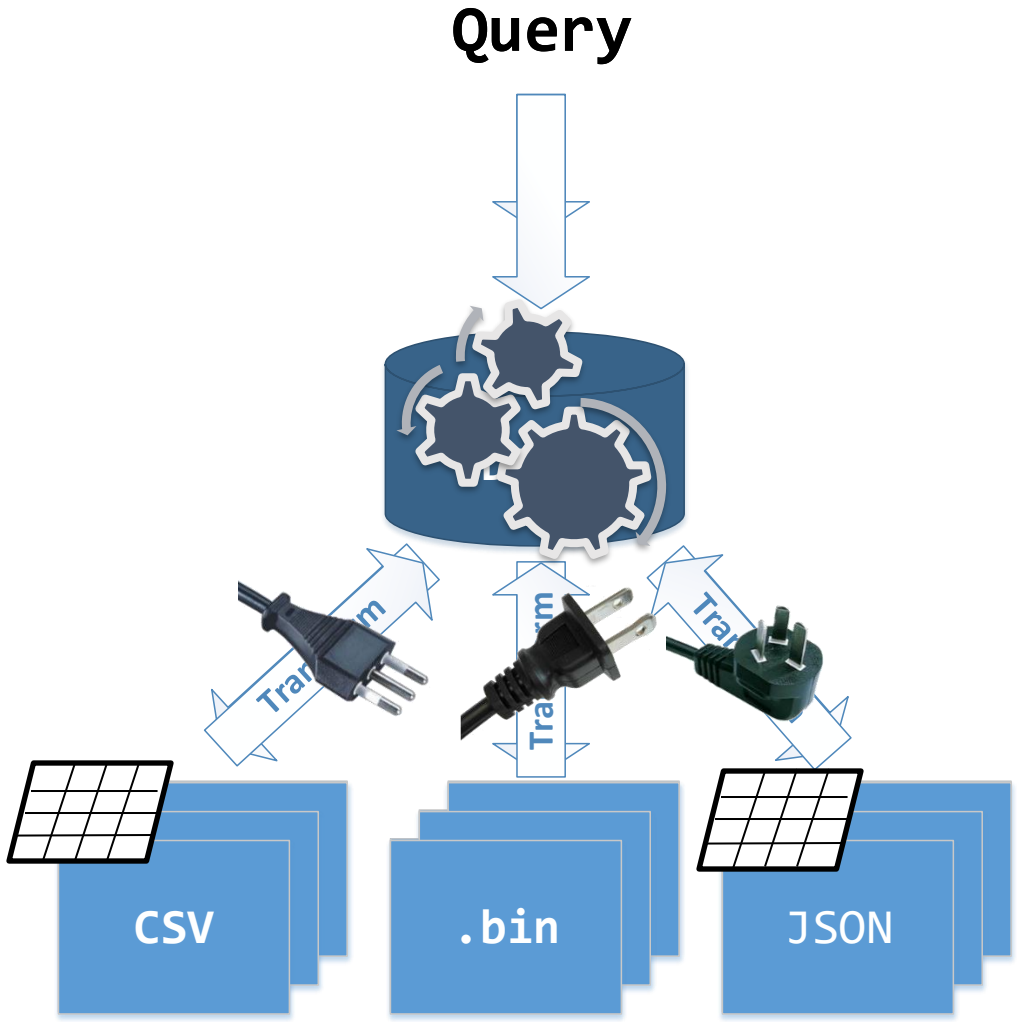


Low-level, error-prone coding

Query compilation

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

Processing over RAW data



Traditional DBMS:
Data adapts to engine

Proteus

Plug-in per data source

Generic-purpose scan operator

Specialize access paths to formats

Adapting to format

- Unroll Columns

```
∀col:
if col needed:
  ...
```

⇨

```
skipField();
skipRest();
```

- Data Types

```
//read field from file
raw = readNextFieldFromFile(file)
switch (schemaDataType[column])
   case IntType: datum =
convertToInteger(raw)
   case FloatType: datum =
convertToFloat(raw)
```

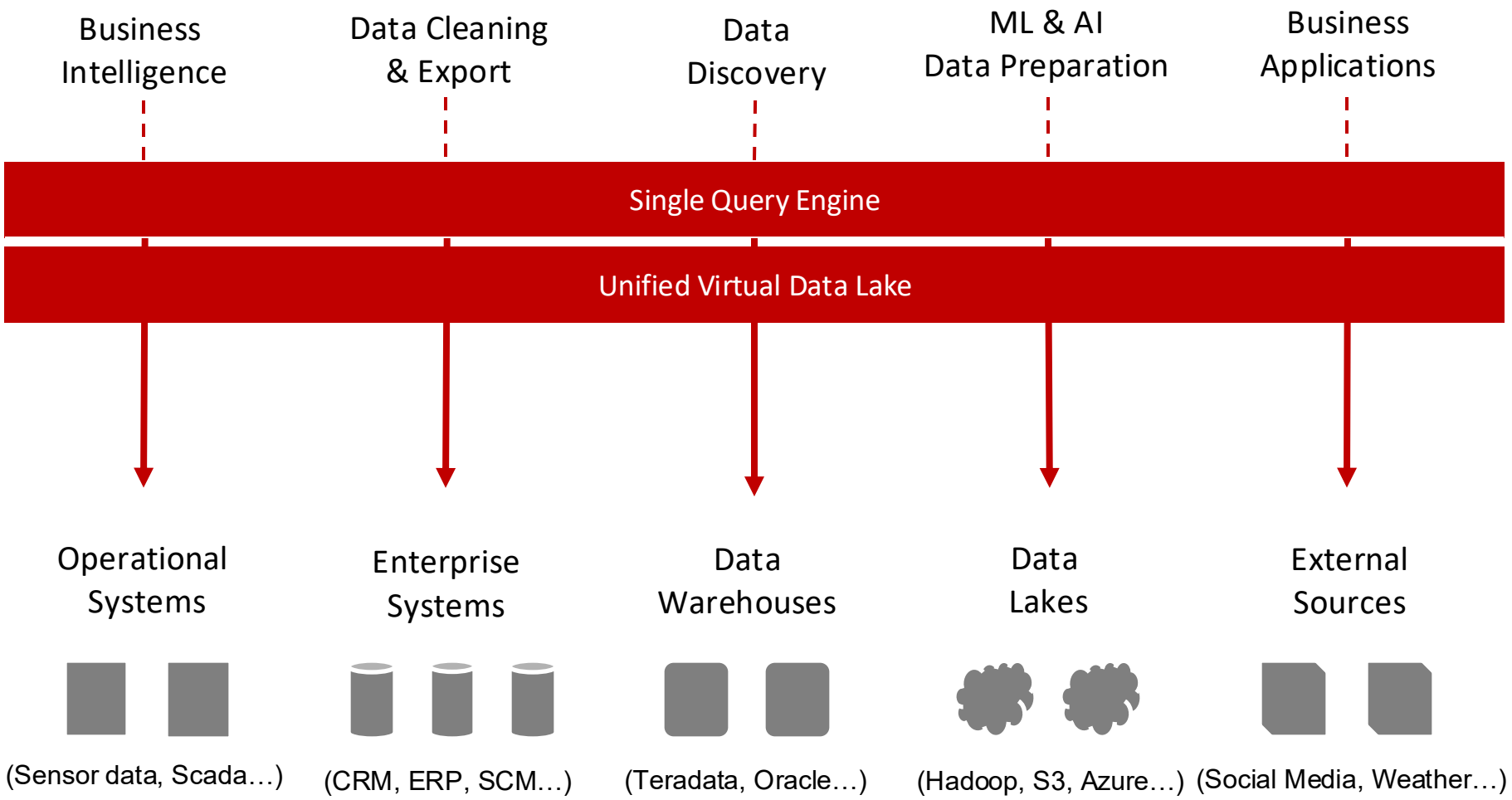
- Free navigation in files

- fieldLength:10
- tupleLength:100
- Need fields 2 & 5 of 2nd row

⇨

```
moveTo(110);
readInt();
moveTo(140);
readFloat();
```

RAW™ NoDB Platform



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)

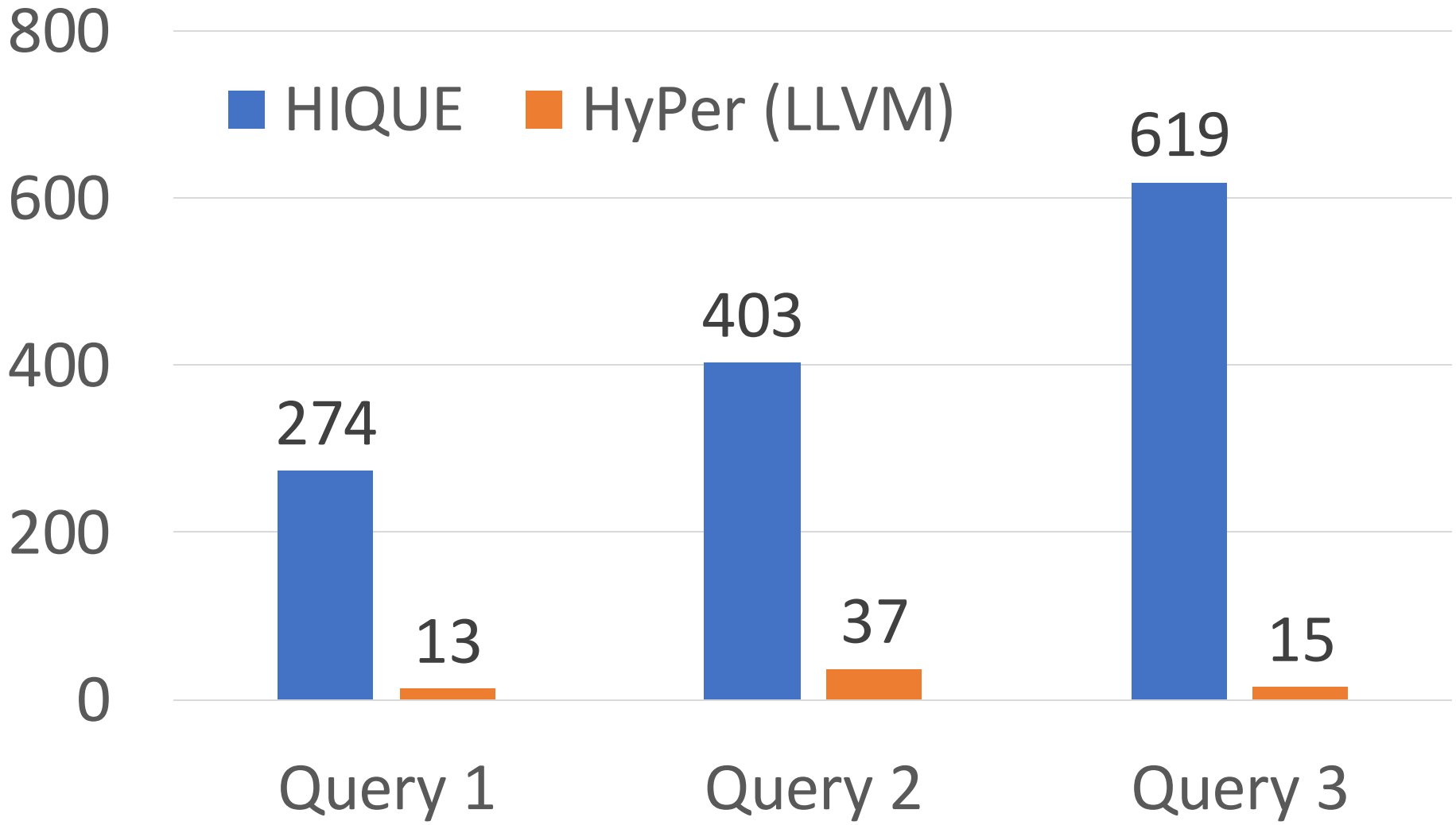
Hybrids do exist!!!

Reading material

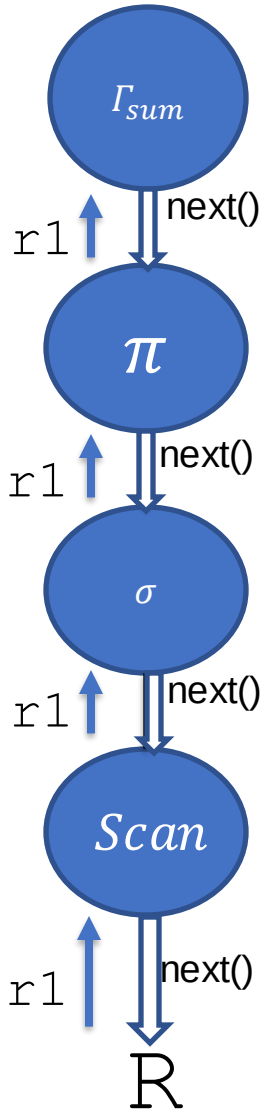
- COW Book chapters 12
- M. Zukowski et al: MonetDB/X100 - A DBMS In The CPU Cache. IEEE 2005: https://ir.cwi.nl/pub/11098/11098B.pdf
- T. Neumann: Efficiently Compiling Efficient Query Plans for Modern Hardware VLDB 2011: https://www.vldb.org/pvldb/vol4/p539-neumann.pdf

Backup Slides

Compilation cost



Volcano iterator model



- Pull-based Interface: open/next/close
- Tuple-at-a-time processing
- Each operator produces a tuple stream

Simple interface, pipelined execution

Volcano CPU costs

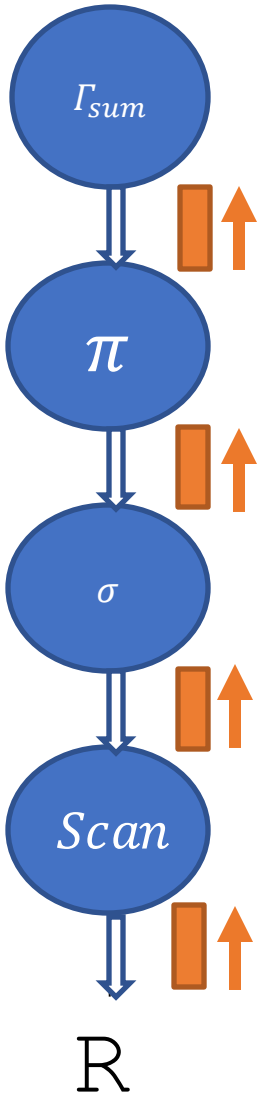
[Graefe]

- Control flow constantly changing
- Many (virtual) function calls
- Branch mispredictions

“Real Work” a fraction of total execution

Vectorized Query Execution

[Zukowski]



- Column-store implementation
- Iterators return blocks

Pros	**Cons**
Fewer next() calls	Materializing costs
Good locality	
Less branching	

Efficient control flow, but wasteful to bandwidth

Generic code is more and slower!

```
1  // loop over pages
2  for (int p = start_page; p <= end_page; p++) {
3   page_struct *page = read_page(p, table);
4   // loop over tuples
5   for (int t = 1; t <= page->num_tuples; t++) {
6    tuple_struct *tuple = read_tuple(t, page);
7    if (!(matches(tuple, predicate_value, predicate_offset)) continue;
8    add_to_result(tuple);
9  }}
```

Listing 3.2: Type-specific table scan-select

```
1  // loop over pages
2  for (int p = start_page; p <= end_page; p++) {
3   page_struct *page = read_page(p, table);
4   // loop over tuples
5   for (int t = 0; t < page->num_tuples; t++) {
6    void *tuple = page->data + t * tuple_size;
7    int *value = tuple + predicate_offset;
8    if (*value != predicate_value) continue;
9    memcpy(..);
10 }}
```

Source:
PhD thesis of
K. Krikellas

Listing 3.3: Naïve nested loops join

```
1  // loop over pages
2  for (int p_R = start_page_R; p_R <= end_page_R; p_R++) {
3   page_struct *page_R = read_page(p_R, R);
4   for (int p_S = start_page_S; p_S <= end_page_S; p_S++) {
5    page_struct *page_S = read_page(p_S, S);
6
7    // loop over tuples
8    for (int t_R = 1; t_R <= page_R->num_tuples; t_R++) {
9     tuple_struct *tuple_R = read_tuple(t_R, page_R) ;
10    for (int t_S = 1; t_S <= page_S->num_tuples; t_S++) {
11     tuple_struct *tuple_S = read_tuple(t_S, page_S);
12     if (!(matches(tuple_R, offset_R, tuple_S, offset_S))) continue;
13     add_to_result(tuple_R, tuple_S);
14 }}}}
```

Listing 3.4: Holistic nested loops join

```
1  // loop over pages
2  for (int p_R = start_page_R; p_R <= end_page_R; p_R++) {
3   page_struct *page_R = read_page(p_R, R);
4   for (int p_S = start_page_S; p_S <= end_page_S; p_S++) {
5    page_struct *page_S = read_page(p_S, S);
6
7    // loop over tuples
8    for (int t_R = 0; t_R < page_R->num_tuples; t_R++) {
9     void *tuple_R = page_R->data + t_R * tuple_size_R;
10    for (int t_S = 0; t_S < page_S->num_tuples; t_S++) {
11     void *tuple_S = page_S->data + t_S * tuple_size_S;
12     int *attr_R = tuple_R + offset_R;
13     int *attr_S = tuple_S + offset_S;
14     if (*attr_R != *attr_S) continue;
15     add_to_result(tuple_R, tuple_S);    /* inlined */
16 }}}}
```

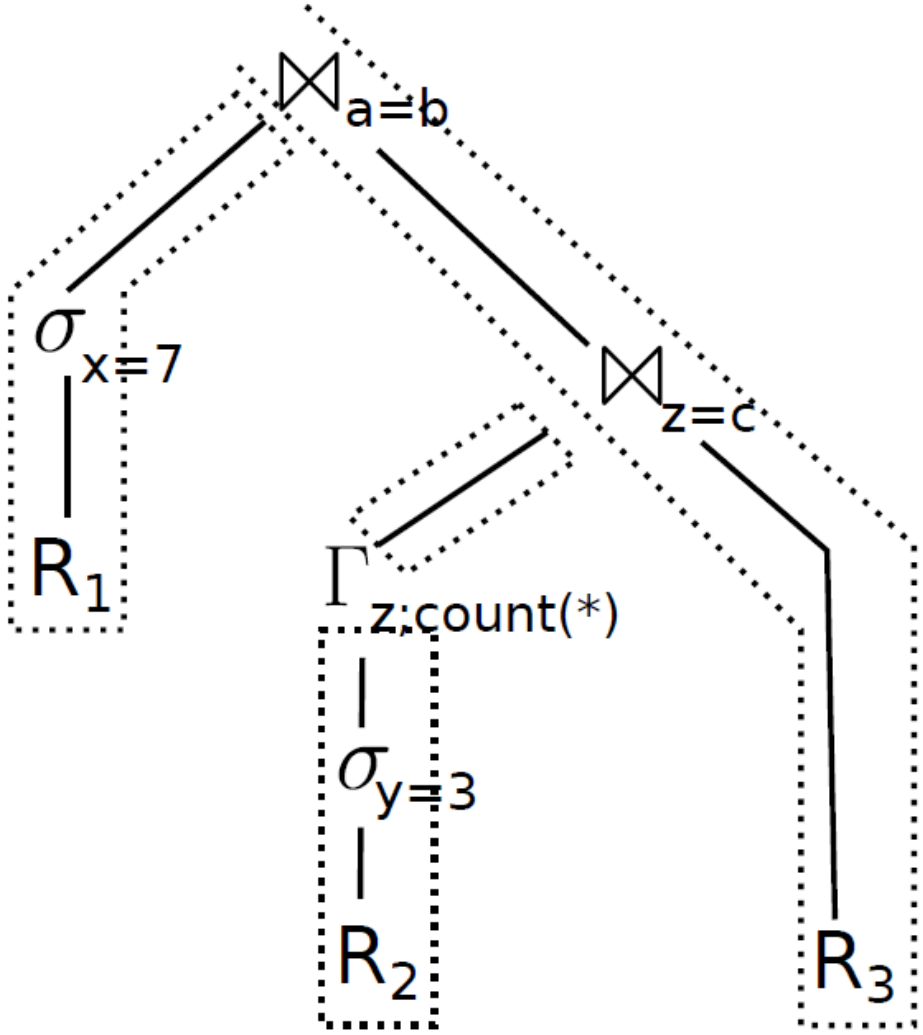
Exercise:

- Compare the two code snippets.
- Which is better? Why?

Source:

PhD thesis of K. Krikellas

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.

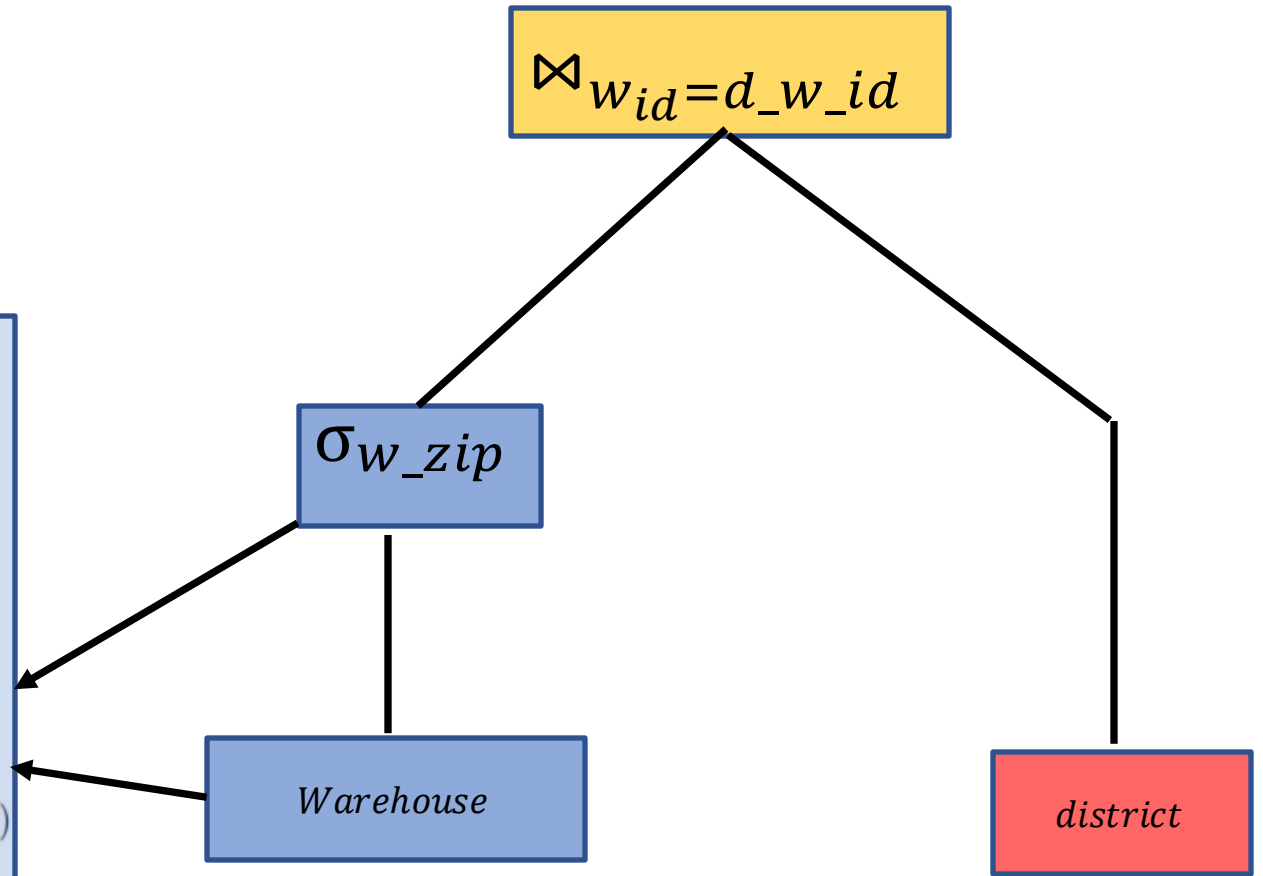
Execute without "spilling data to memory"

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
```

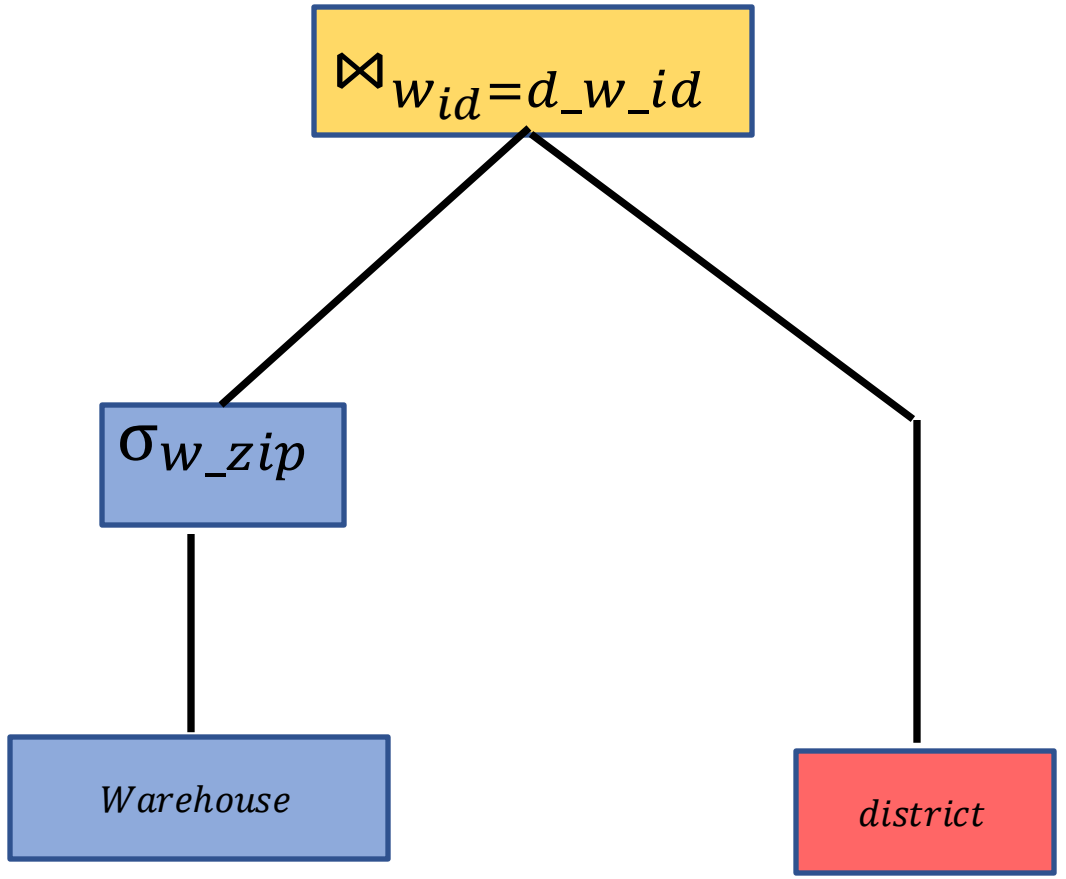


(more)

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

```
%30 = call i64 @llvm.x86.sse42.crc64.64(i64 0, i64 %29)
%31 = shl i64 %30, 32
%32 = call i8* @_ZN5hyper9HashTable15storeInputTupleEmj(%"hyper::
    HashTable"* %11, i64 %31, i32 4)
%33 = bitcast i8* %32 to i32*
store i32 %w_id, i32* %33, align 1
br label %cont3

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
```

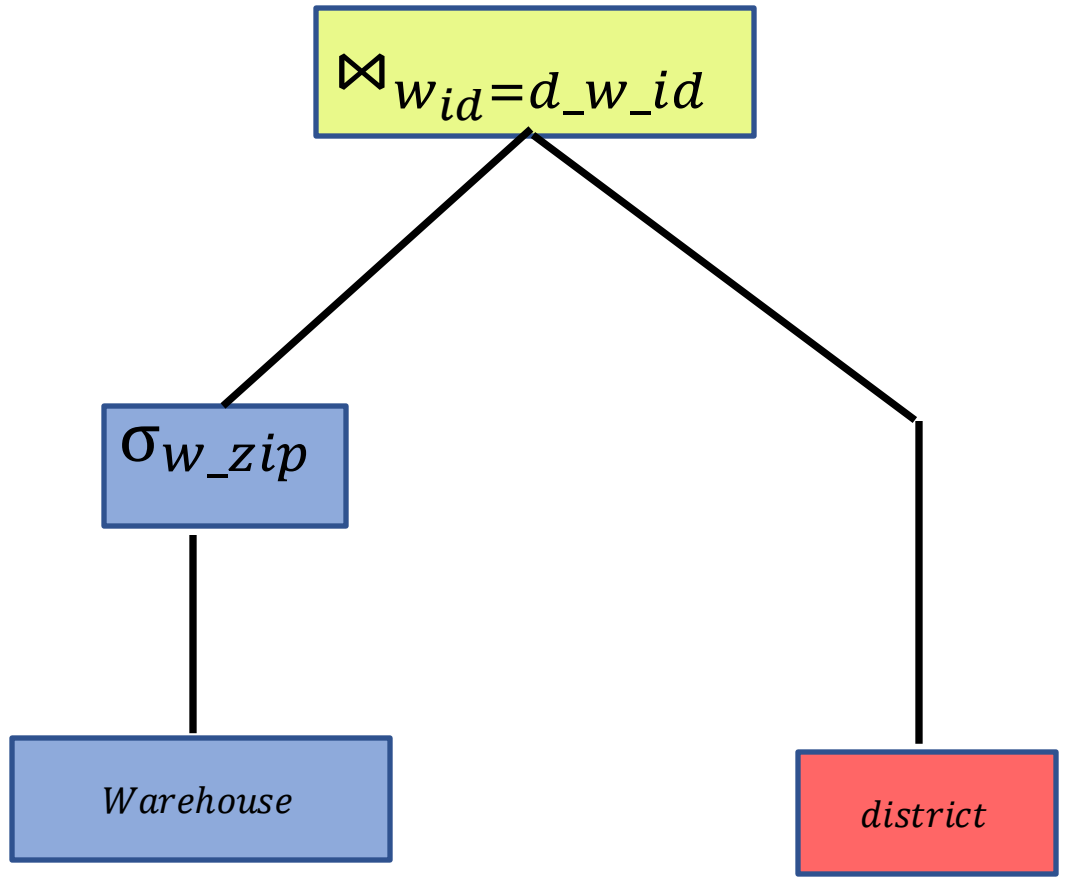
```
scanBody5:
  %tid9 = phi i64 [ 0, %scanDone ], [ %58, %loopDone ]
  %36 = getelementptr i32* %21, i64 %tid9
  %d_w_id = load i32* %36, align 4
  %37 = getelementptr i64* %24, i64 %tid9
  %d_tax = load i64* %37, align 8
  %38 = zext i32 %d_w_id to i64
  %39 = call i64 @llvm.x86.sse42.crc64.64(i64 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
```



(Even more!)

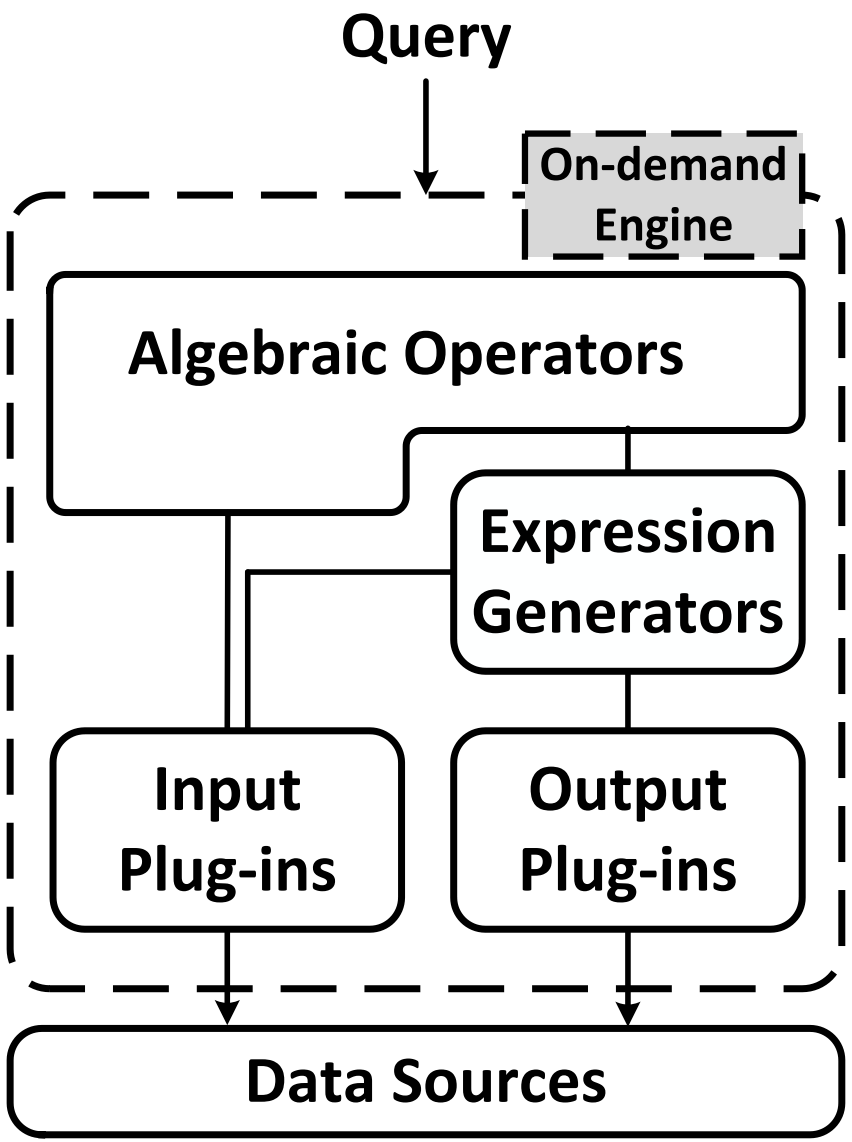
```
select d_tax from warehouse, district where
    w_id=d_w_id and w_zip='…'
```

```
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

From query plan to code



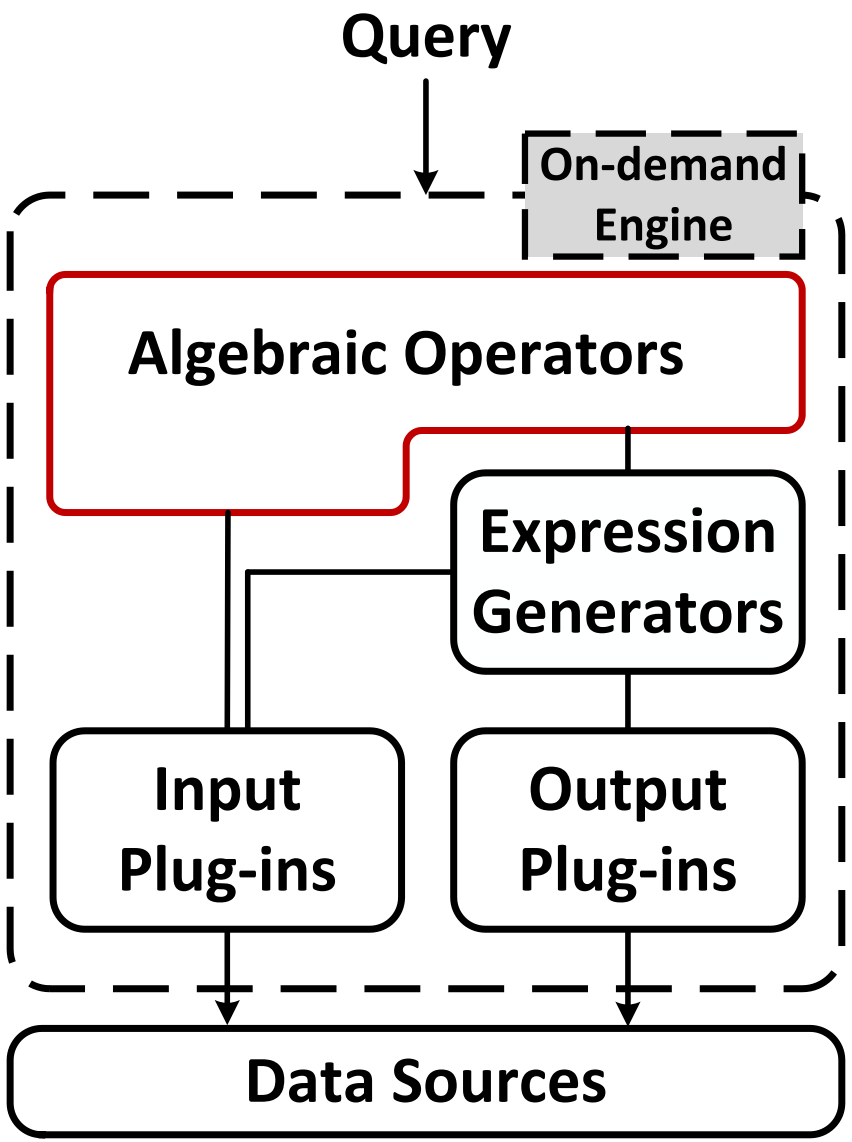

```
SELECT COUNT(*)
FROM Sailors
WHERE age < 20
```

Δ^{sum}

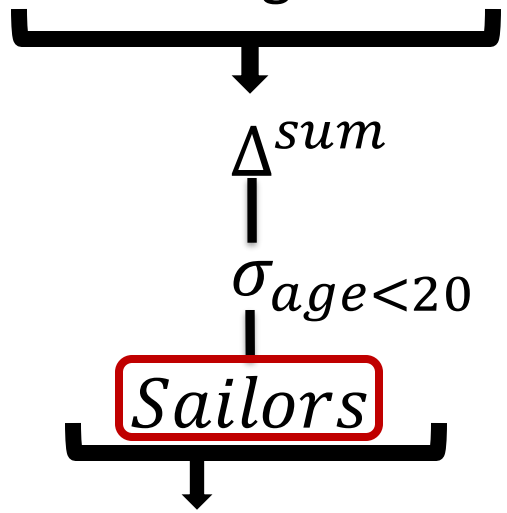
$\sigma_{age<20}$

$Sailors$

From query plan to code

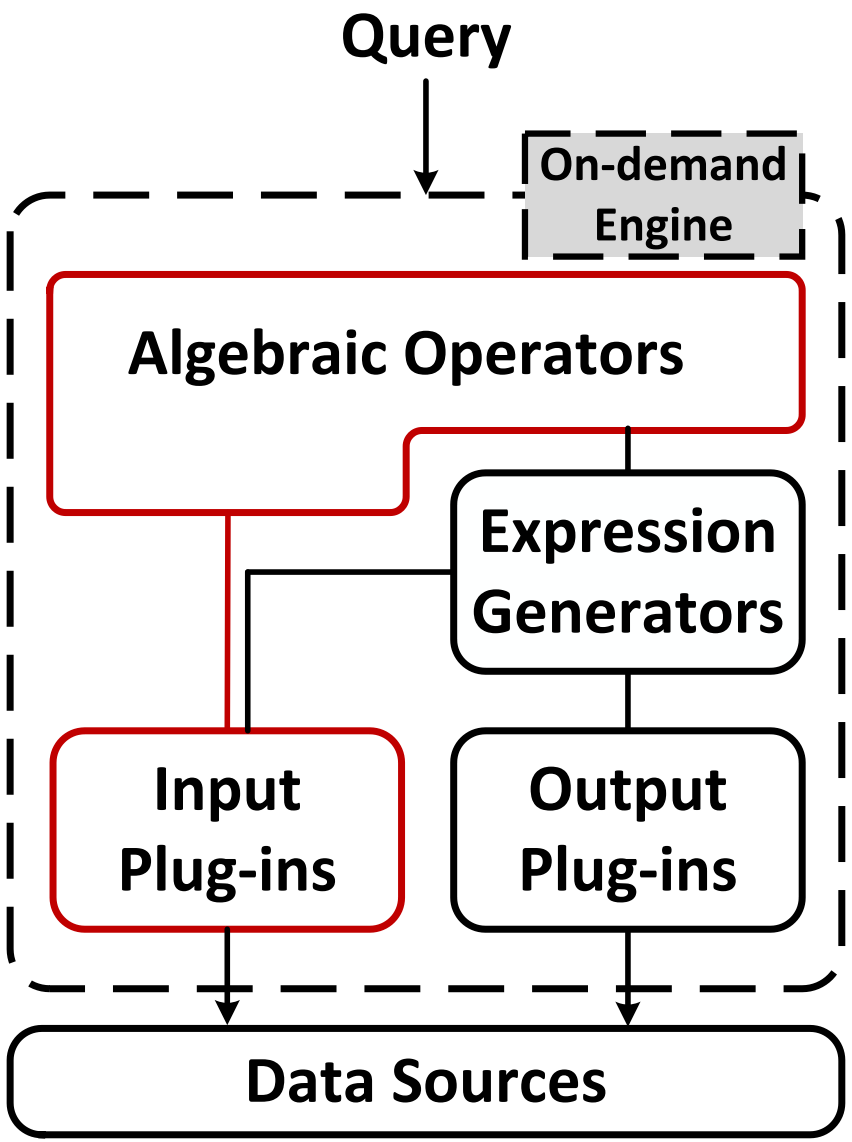



```
SELECT COUNT(*)
FROM Sailors
WHERE age < 20
```




```
while() {


}
```

From query plan to code




```
SELECT COUNT(*)
FROM Sailors
WHERE age < 20
```

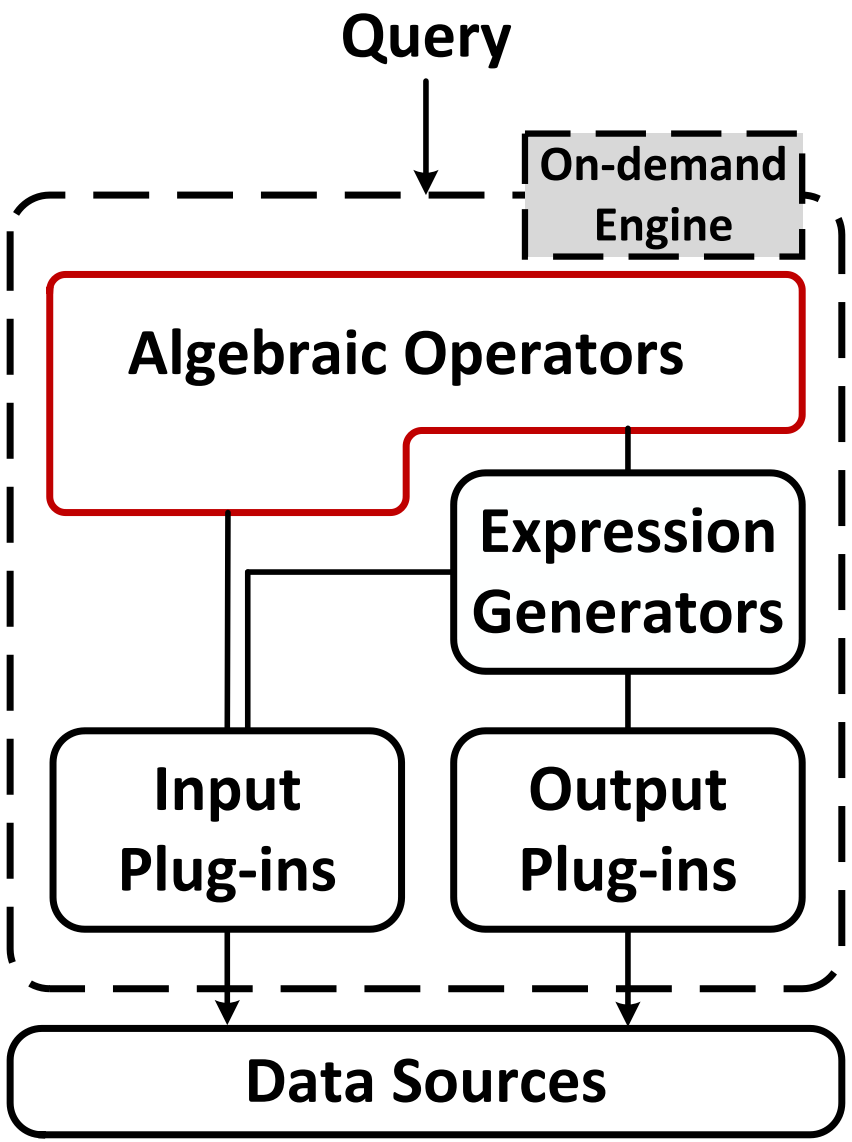
Δ^{sum}

$\sigma_{age<20}$

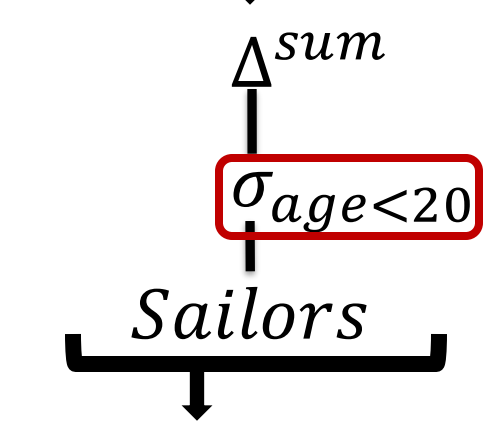
$Sailors$

```
while(!eof(Sailors)) {
  read(age)


}
```

From query plan to code

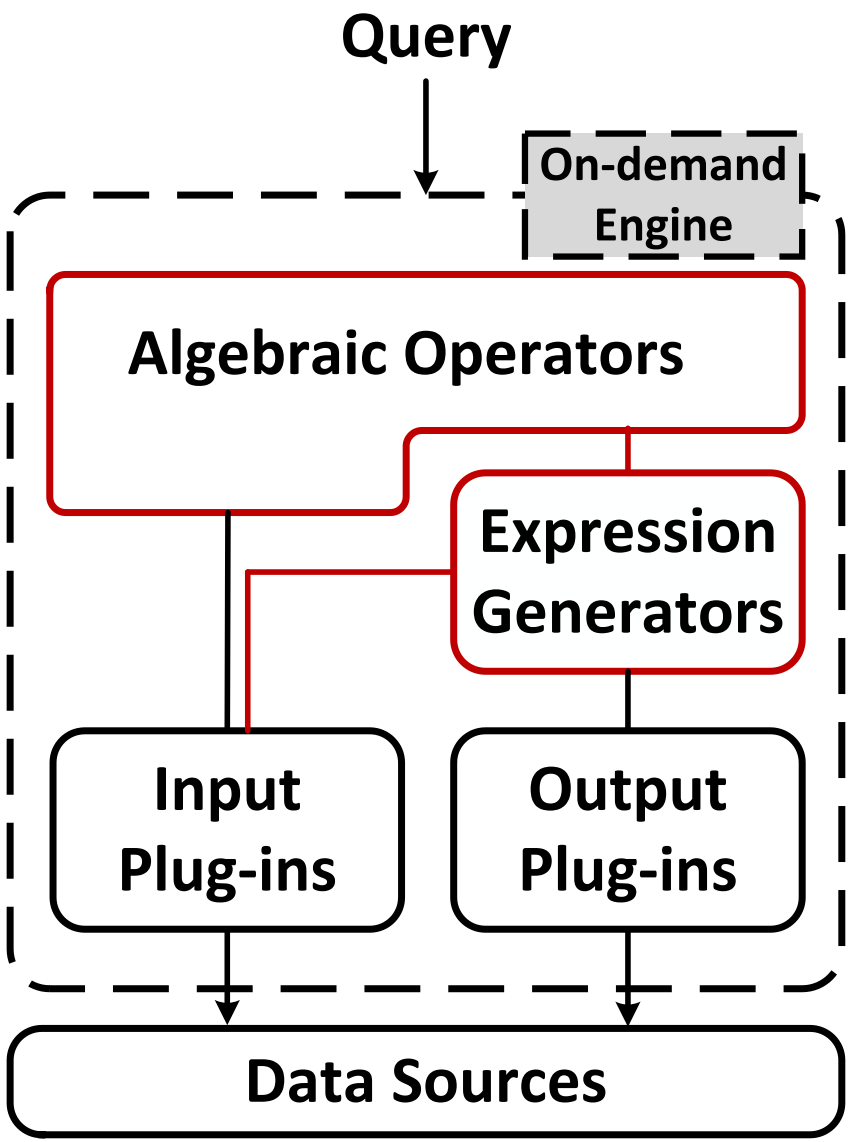



```
SELECT COUNT(*)
FROM Sailors
WHERE age < 20
```




```
while(!eof(Sailors)) {
  read(age)
  if() {

  }
}
```

From query plan to code




```
SELECT COUNT(*)
FROM Sailors
WHERE age < 20
```

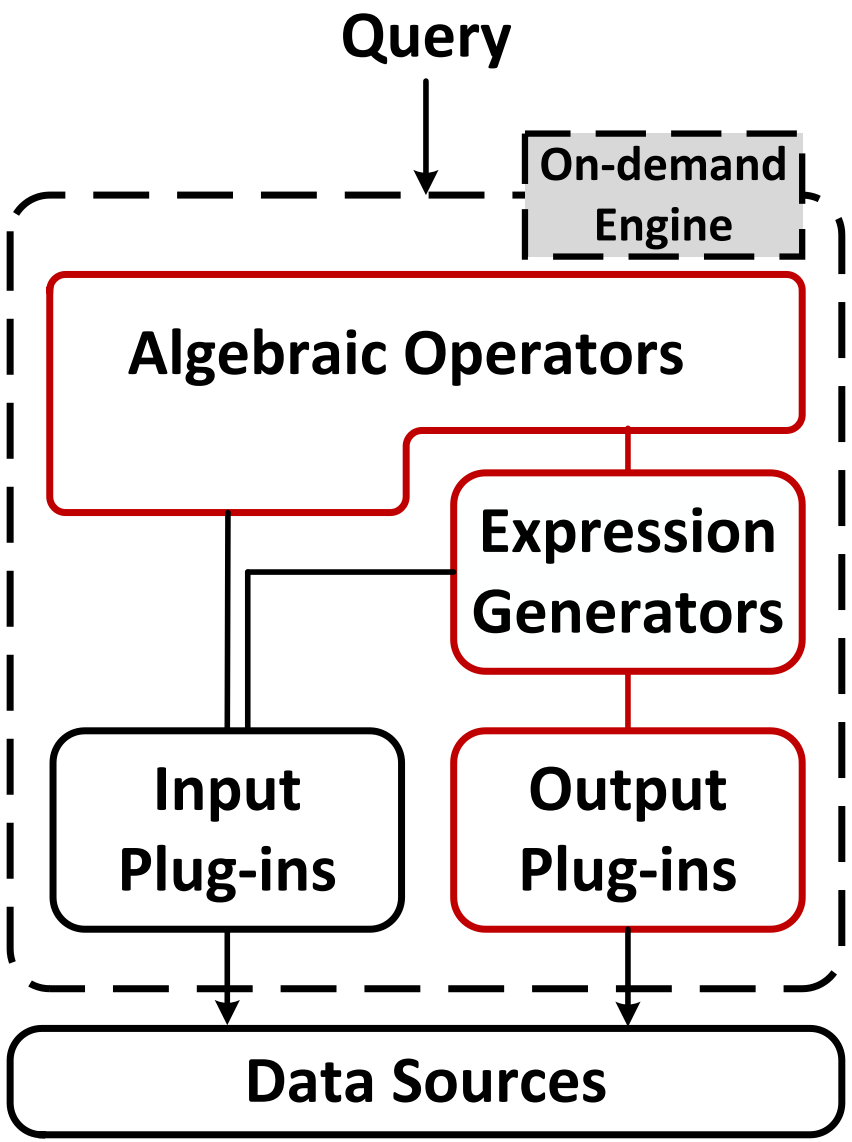
Δ^{sum}

$\sigma_{age<20}$

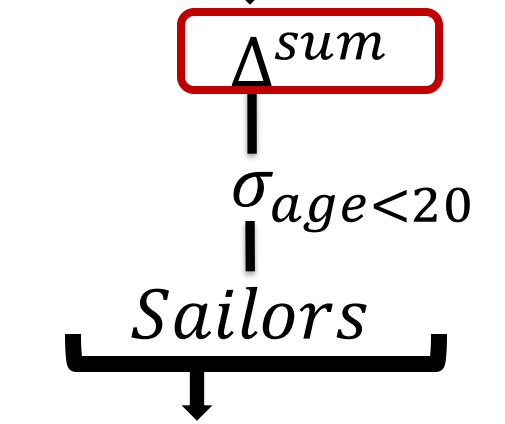
$Sailors$

```
while(!eof(Sailors)) {
  read(age)
  if(eval(age < 20)) {

  }
}
```

From query plan to code




```
SELECT COUNT(*)
FROM Sailors
WHERE age < 20
```




```
while(!eof(Sailors)) {
  read(age)
  if(eval(age < 20)) {
    sum += 1
  }
}
return out(sum,json)
```