JIT In The Wild: What Databases Do To Speed Up Your Queries

Alexey Goncharuk,

Querify Labs, CTO

04 July 2023

Co-organizer

Yandex

About

- CTO at Querify Labs
  - We help technology companies build new databases and data management systems
- Developed Apache Ignite for nine years
- At Querify Labs
  - Query Optimizers
  - SQL Execution Runtime

What Is JIT?

Co-organizer

Yandex

JIT: Quick Intro

- Execute code that is generated during the program execution
- At runtime there is additional information that is not available at compile time
- Supposedly, we can generate more optimal code

JIT Intro: In native environment

- Write generated machine code into a memory buffer
- Jump to the generated code and execute

```
void (*optimized)() =
    (void (*)())generateFastFunction();


optimized();
```

JIT Intro: In managed environment

- Stricter control over the executed code
- Generate Virtual Machine instructions instead
- Java: **`ClassLoader.defineClass()`**
- C#: **`Assembly.Load()`**

Jit Intro: Opportunities

Code Specialization

- Use runtime knowledge to
  - Avoid branching
  - Eliminate unnecessary boxing
  - Get rid of lookup tables
  - Eliminate dead code

Jit Intro: Approaches

- Transpilation
  - Generate code from templates
  - Invoke a regular compiler to generate binary code
- Binary code generation
  - Generate machine code directly or
  - Use tool Intermediate Representation to describe control flow

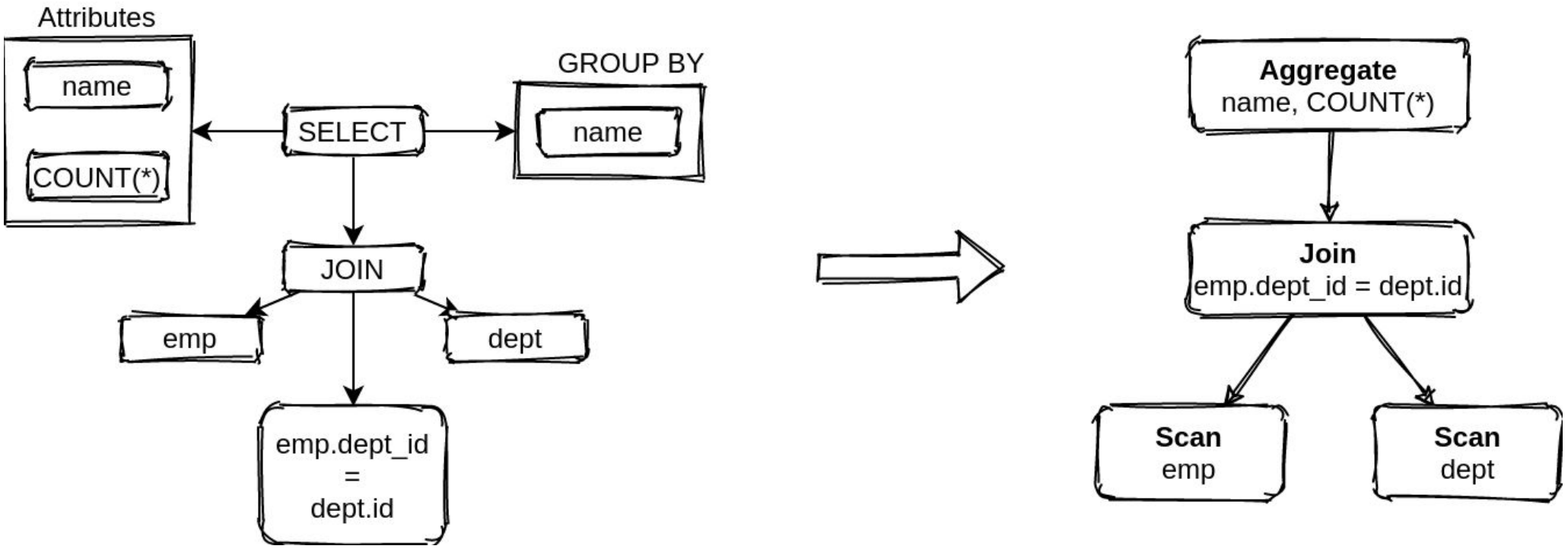
JIT For Relational Operators

Co-organizer

Yandex

Relational Operators

- Query is represented as an operator tree
- Each operator has a well-defined and narrow semantics



Operator Types

Operator	Description
Scan	Scan a data source
Project	Transform tuple attributes
Filter	Filter rows according to a predicate
Sort	ORDER BY / LIMIT / OFFSET
Aggregate	Aggregate operator
Window	Window aggregation
Join	2-way join
Union/Minus/Intersect	N-way set operators

Operators Tree

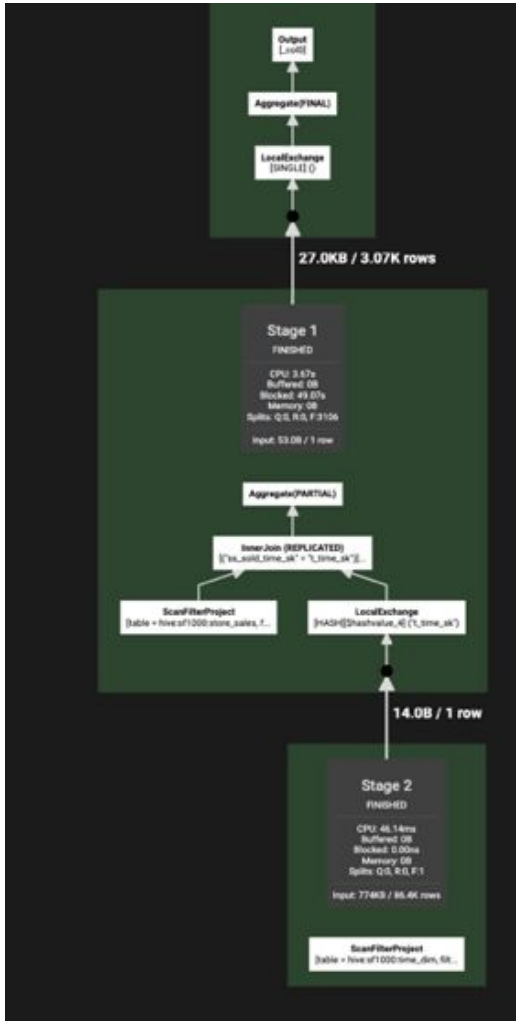
PG: **`EXPLAIN / EXPLAIN ANALYZE`**

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Merge Join  (cost=0.56..292.65 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28 rows=101 width=244)
         Filter: (unique1 < 100)
   ->  Index Scan using onek_unique2 on onek t2  (cost=0.28..224.79 rows=1000 width=244)
```

Operators Tree

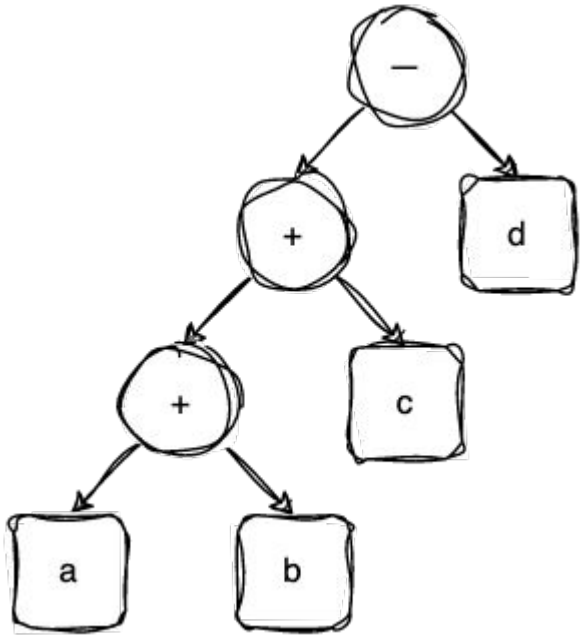
Trino: **`EXPLAIN`** and Live Query Plan

```
                                                Query Plan
-----------------------------------------------------------------------------------------
 Trino version: version
 Output[regionkey, _col1]
 |   Layout: [regionkey:bigint, count:bigint]
 |   Estimates: {rows: ? (?), cpu: ?, memory: ?, network: ?}
 |   _col1 := count
 └─ RemoteExchange[GATHER]
    |   Layout: [regionkey:bigint, count:bigint]
    |   Estimates: {rows: ? (?), cpu: ?, memory: ?, network: ?}
    └─ Aggregate(FINAL)[regionkey]
       |   Layout: [regionkey:bigint, count:bigint]
       |   Estimates: {rows: ? (?), cpu: ?, memory: ?, network: ?}
       |   count := count("count_8")
       └─ LocalExchange[HASH][$hashvalue] ("regionkey")
          |   Layout: [regionkey:bigint, count_8:bigint, $hashvalue:bigint]
          |   Estimates: {rows: ? (?), cpu: ?, memory: ?, network: ?}
          └─ RemoteExchange[REPARTITION][$hashvalue_9]
             |   Layout: [regionkey:bigint, count_8:bigint, $hashvalue_9:bigint]
             |   Estimates: {rows: ? (?), cpu: ?, memory: ?, network: ?}
             └─ Project[]
                |   Layout: [regionkey:bigint, count_8:bigint, $hashvalue_10:bigi
                |   Estimates: {rows: ? (?), cpu: ?, memory: ?, network: ?}
                |   $hashvalue_10 := "combine_hash"(bigint '0', COALESCE("$operat
                └─ Aggregate(PARTIAL)[regionkey]
                   |   Layout: [regionkey:bigint, count_8:bigint]
                   |   count_8 := count(*)
                   └─ TableScan[tpch:nation:sf0.01]
                          Layout: [regionkey:bigint]
                          Estimates: {rows: 25 (225B), cpu: 225, memory: 0B, netw
                          regionkey := tpch:regionkey
```



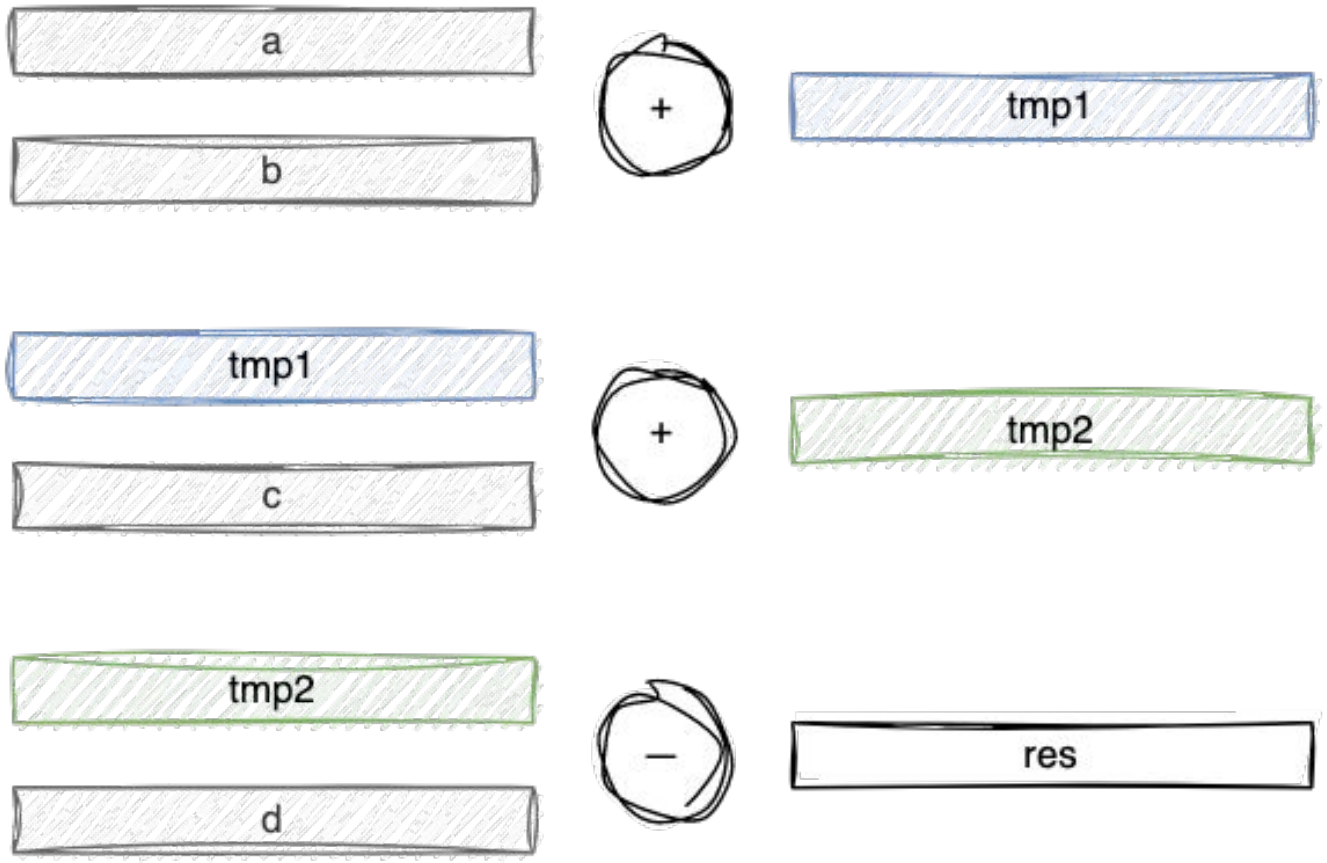
JIT: Project / Filter


```
SELECT a + b + c - d FROM table
WHERE e > 0
```



JIT: Project / Filter

Vectorized execution: intermediate materializations



Number of extra materializations is proportional to the expression tree depth

JIT: Project / Filter


```
for (auto row : batch) {
    int32_t a = c0.value(row);
    int32_t b = c1.value(row);
    int32_t c = c2.value(row);
    int32_t d = c3.value(row);

    auto res = a + b + c - d;
    out0.append(res);
}
```

- Intermediate results are placed in registers
- No explicit type dispatch
- Nullability check is eliminated
- Loop may be SIMD

JIT: Hash Aggregation / Hash Join


```
SELECT sum(a), avg(b) FROM table
    GROUP BY c, d, e
```

JIT: Hash Aggregation / Hash Join


```
SELECT sum(a), avg(b) FROM table
GROUP BY c, d, e
```

- Core data structure of an aggregation is HashTable
- Entry for interpreted hash table is generic:
- Hash calculation
  - Explicit function call: type dispatch, expensive
  - Vectorized hash calculation: intermediate materializations
- HashTable slot storage
  - Generic record: hard to implement open addressing hash table

JIT: Hash Aggregation / Hash Join


```
struct GroupKey {
    int32_t c;
    int32_t d;
    int32_t e;
}

HashTable<GroupKey> table;
```

- Inlineable hash code / comparison
- Precise control over the table layout

JIT: Sorting / Merge Join


```
SELECT a, b, c FROM table
ORDER BY a, b, c
```

JIT: Sorting / Merge Join


```
SELECT a, b, c FROM table
ORDER BY a, b, c
```

- One of the main operations is tuple comparison
- Comparison function must be generic to handle arbitrary tuple type
- Code specialization
  - More opportunities for inlining and vectorization
  - No explicit type dispatch

Practical Considerations

Co-organizer

Yandex

Transpilation vs Compilation

Transpilation

- Generate regular code
- Easy to debug
- Very quick to bootstrap
- Large compilation time

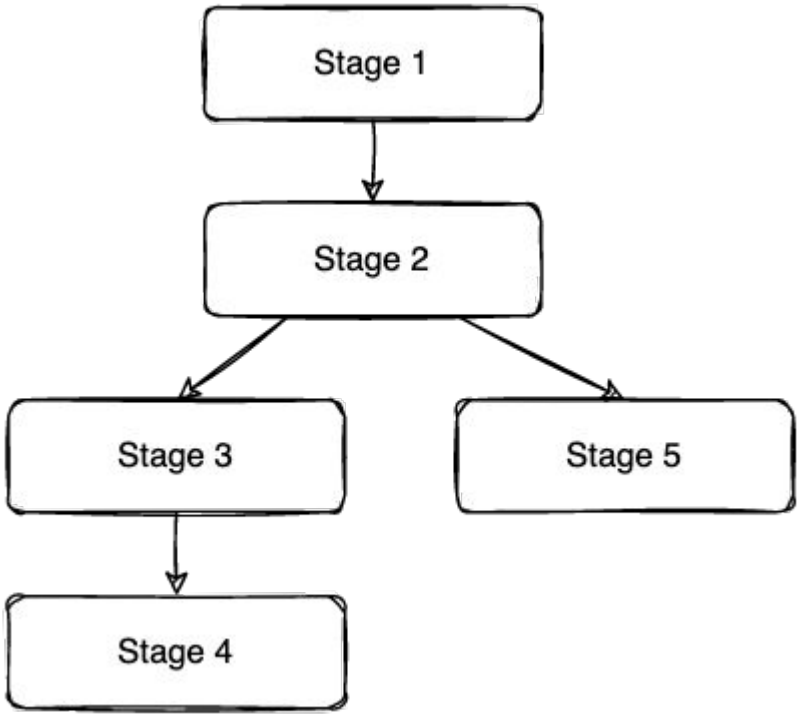
Intermediate Representation (IR) Generation

- Build IR graphs for operator logic
- Hard to read and debug
- Significantly larger times for bootstrap
- Quicker compilation time
- Precise control over binary code generation

Dealing With Compilation Latency

On-Stack Operator Replacement

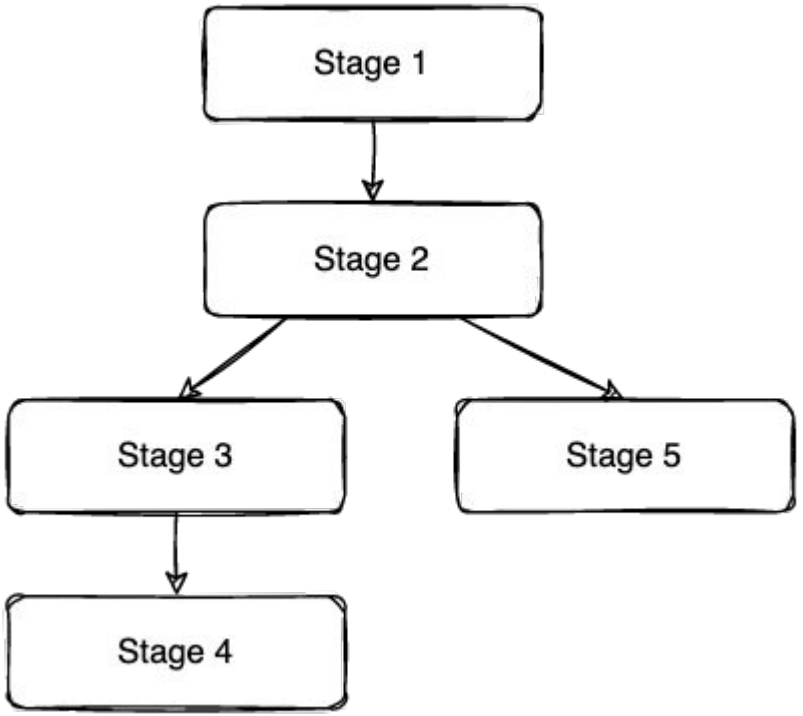
- Streaming operators and pipeline breakers



Dealing With Compilation Latency

On-Stack Operator Replacement

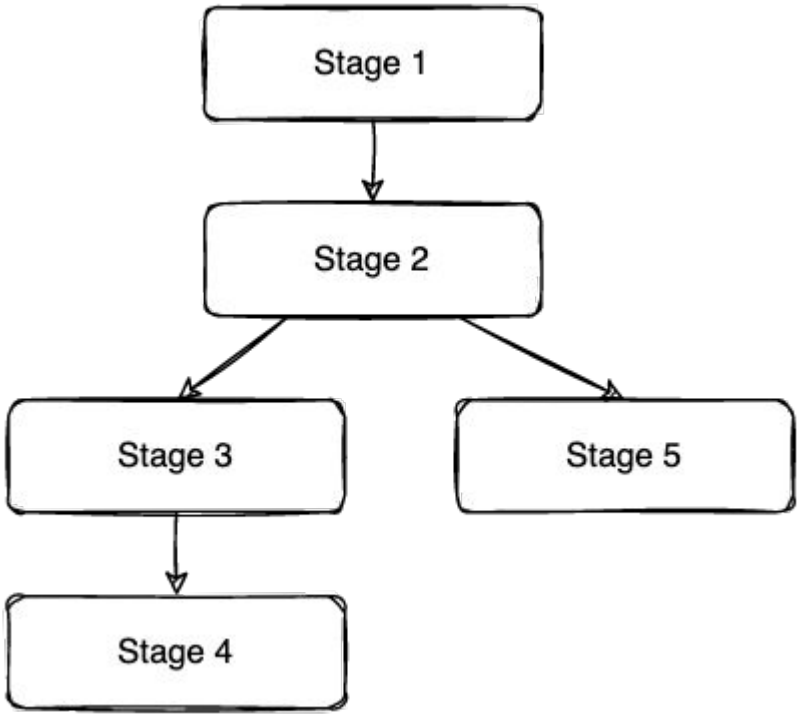
- Streaming operators and pipeline breakers
- Breakers are stages boundaries



Dealing With Compilation Latency

On-Stack Operator Replacement

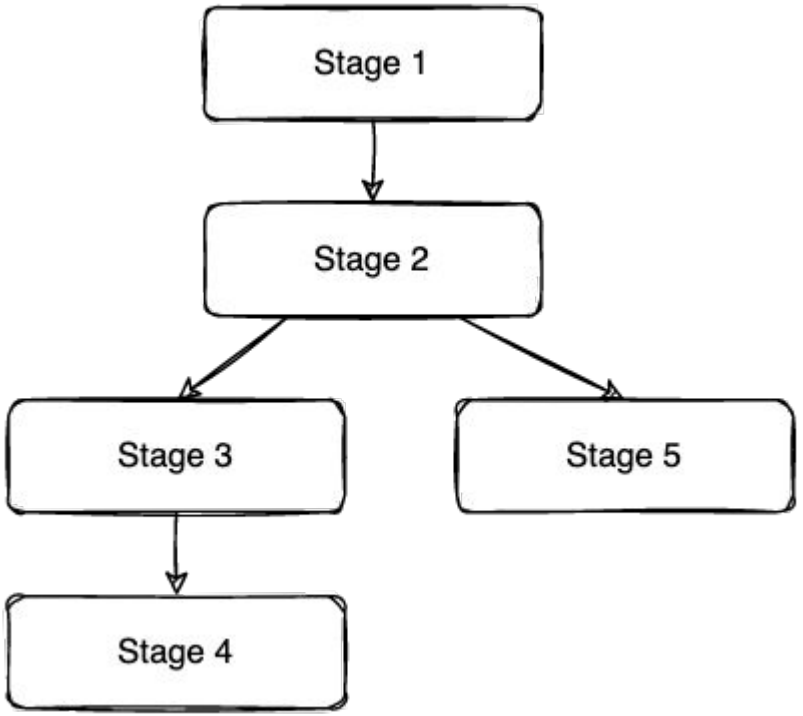
- Streaming operators and pipeline breakers
- Breakers are stages boundaries
- Streaming operators are either fully stateless or may drop state (project, filter, pre-aggregation)



Dealing With Compilation Latency

On-Stack Operator Replacement

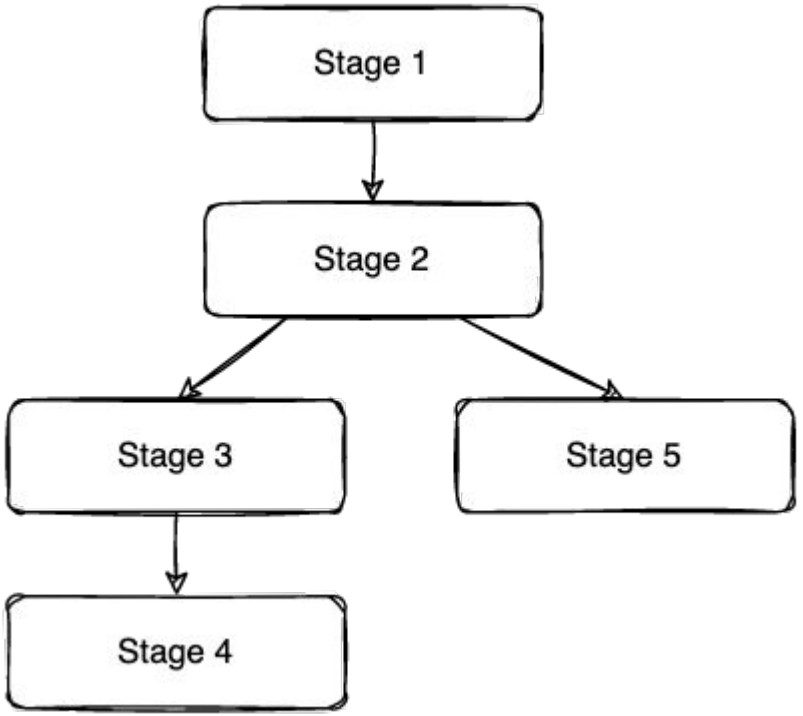
- Streaming operators and pipeline breakers
- Breakers are stages boundaries
- Streaming operators are either fully stateless or may drop state (project, filter, pre-aggregation)
- Prioritize compilation of high-cardinality inputs and top stages



Dealing With Compilation Latency

On-Stack Operator Replacement

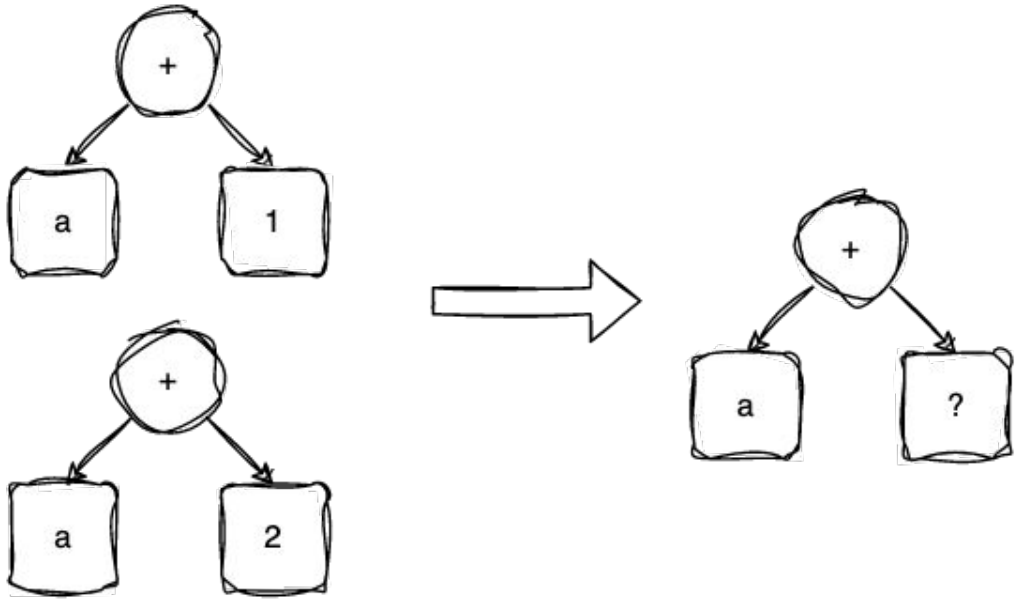
- Need to maintain two sets of operator implementations



Dealing With Compilation Latency

Compilation Cache

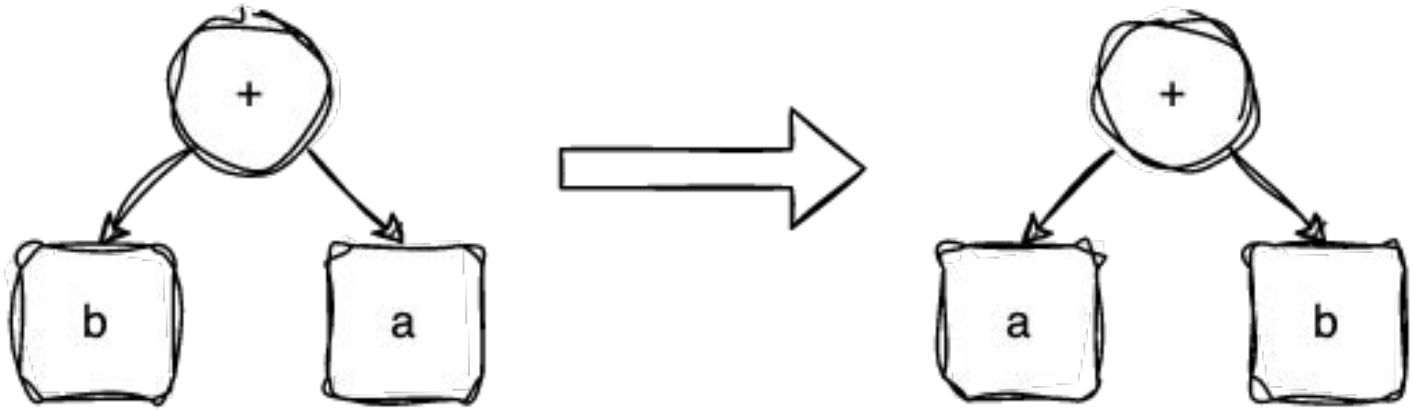
- Use operator descriptor / code as cache key
- Extract constant literals to parameters
  - Projects of a + 1 and a + 2 will use the same code



Dealing With Compilation Latency

Compilation Cache

- Best-effort normalization
  - Projects a + b and b + a will use the same code

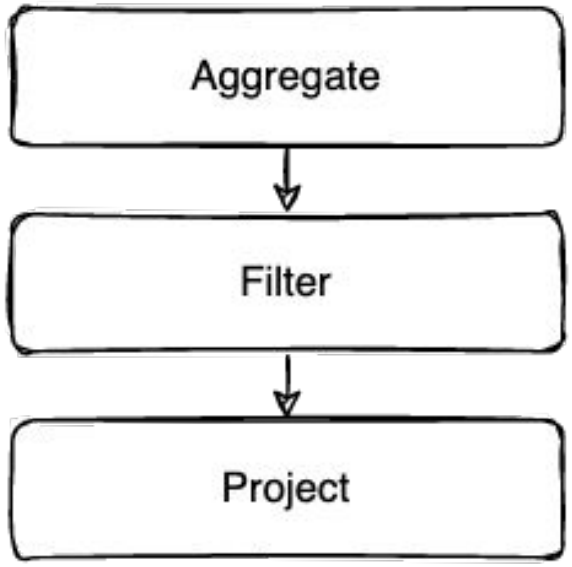


Dealing With Compilation Latency

Compilation Cache

- Use persistent BLOB storage to keep cache across restarts
- Scaling system
  - Move compiler out of execution process

Operator Fusion




```
for (auto row : batch) {
  auto a = c0.value(row);
  if (a > 0) {
    auto b = c1.value(row) + a;
    ht[GroupKey(a, b)]++;
  }
}
```

Operator Fusion

- Decreases chances for cache reuse across queries
- Lower observability
  - Can expose only fused operator statistics
  - Does not map directly to the physical plan
- May drop opportunity for explicit SIMD-ification
- Tighter loop, more opportunities for compiler optimization

JIT Observability

- Zero-overhead (when disabled) arbitrary performance counters
- Emit per-operator symbols for runtime profiling
- Per-operator compilation allows for easier debugging and verification

Tooling

Co-organizer

Yandex

Integration With Arbitrary Projects

- LLVM (C++)
  - Infrastructure for building compilers (including JIT)
  - Multiple front-ends and back-ends
  - Can be used for both IR and transpilation approaches
- ASM (Java)
  - Low-level Java bytecode assembler
- Janino (Java)
  - Alternative Java compiler

Thank You!

High Load ++ Serbia

Co-organizer

Yandex