FLINK FORWARD # ASIA 2022 ONLINE | 实时即未来

How to Write Fast Flink SQL

贺小令 | 阿里巴巴高级技术专家、Apache Flink Committer

01 Flink SQL Insight

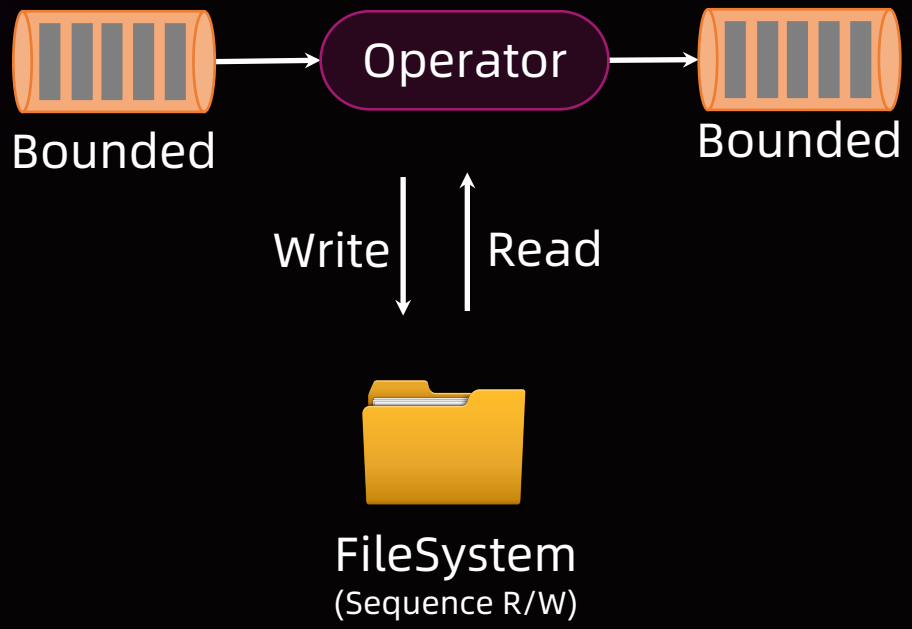
02 Best Practices

03 Future Works

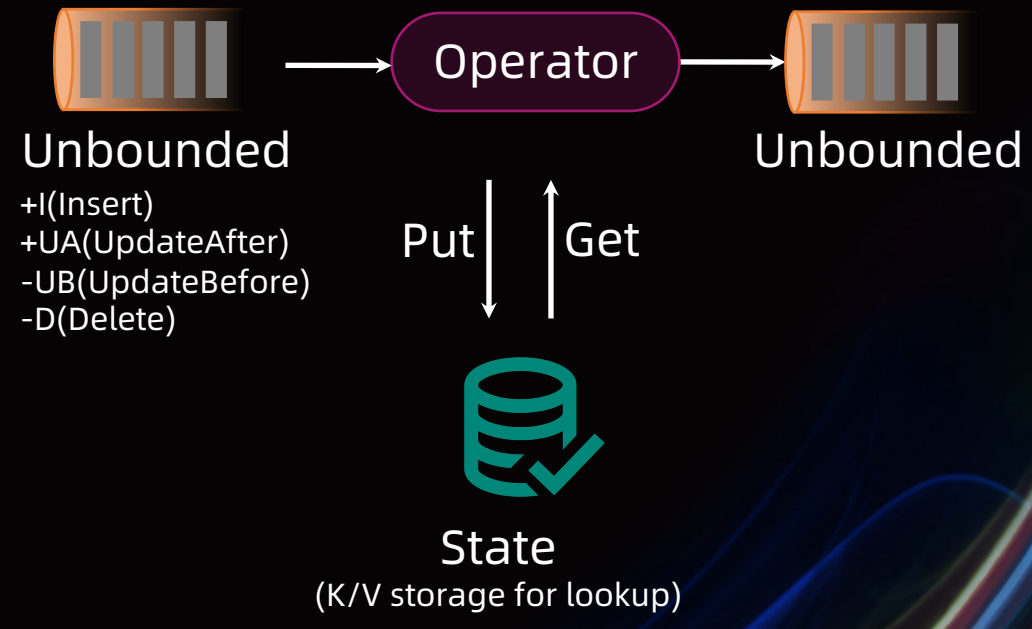
01 Flink SQL Insight

Flink Operator Execution Mode

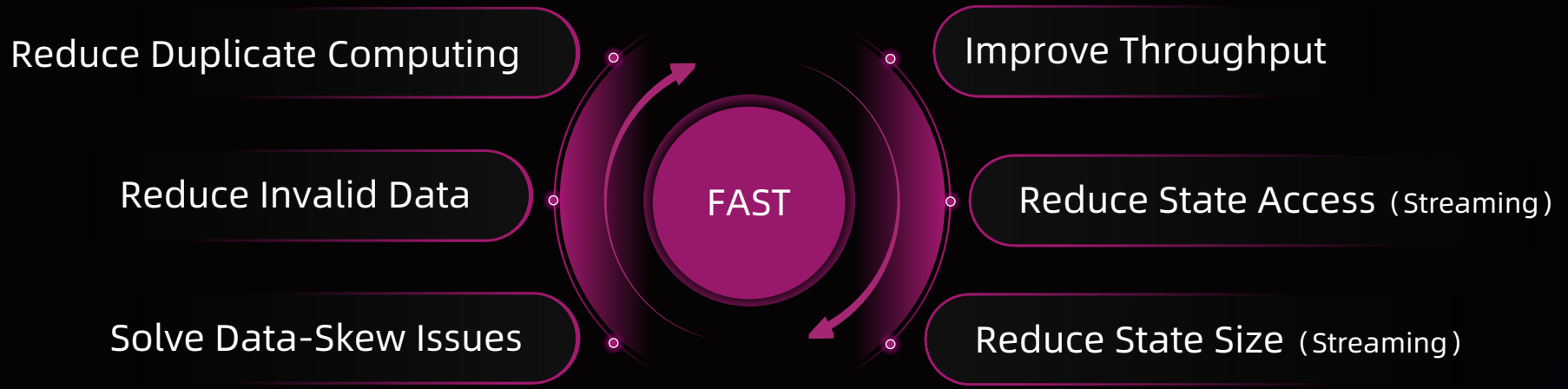
Batch



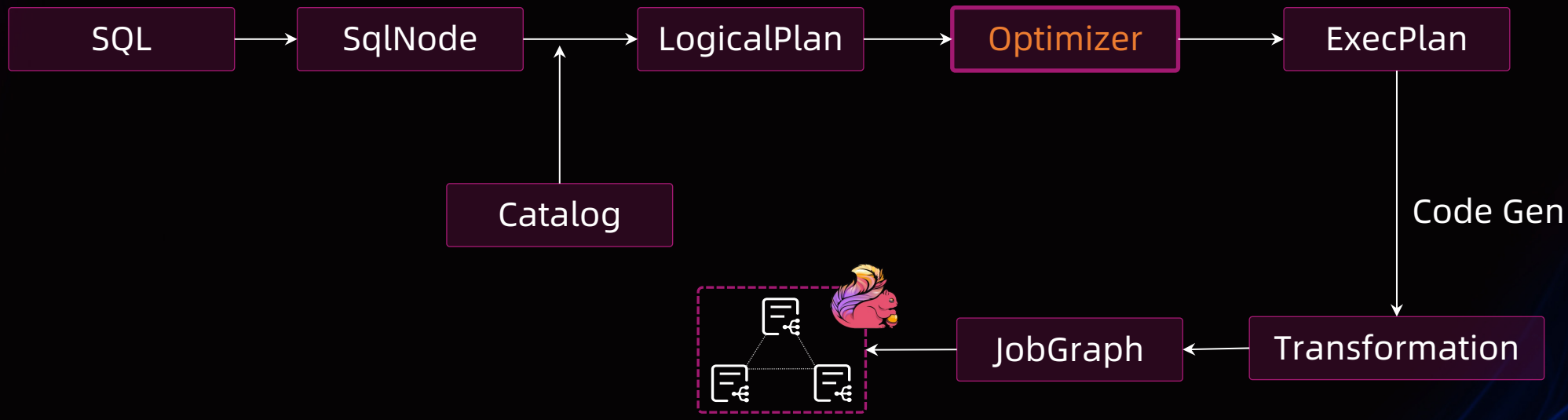
Streaming



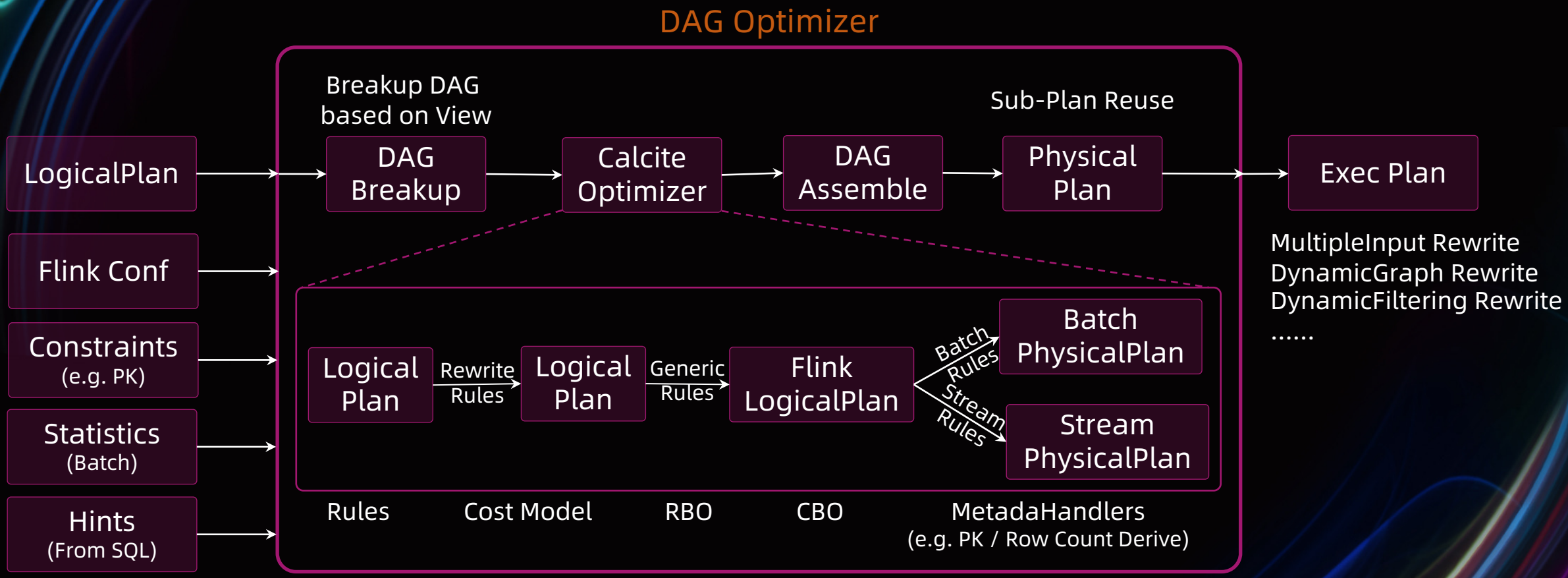
How To Make Flink Job Run Fast



Flink SQL Workflow



Insight Optimizer



02 Best Practices

Sub-Plan Reuse

```
insert into sink1 select * from my_table where a > 10;
insert into sink2 select * from my_table where a > 10 and b < 100;
```

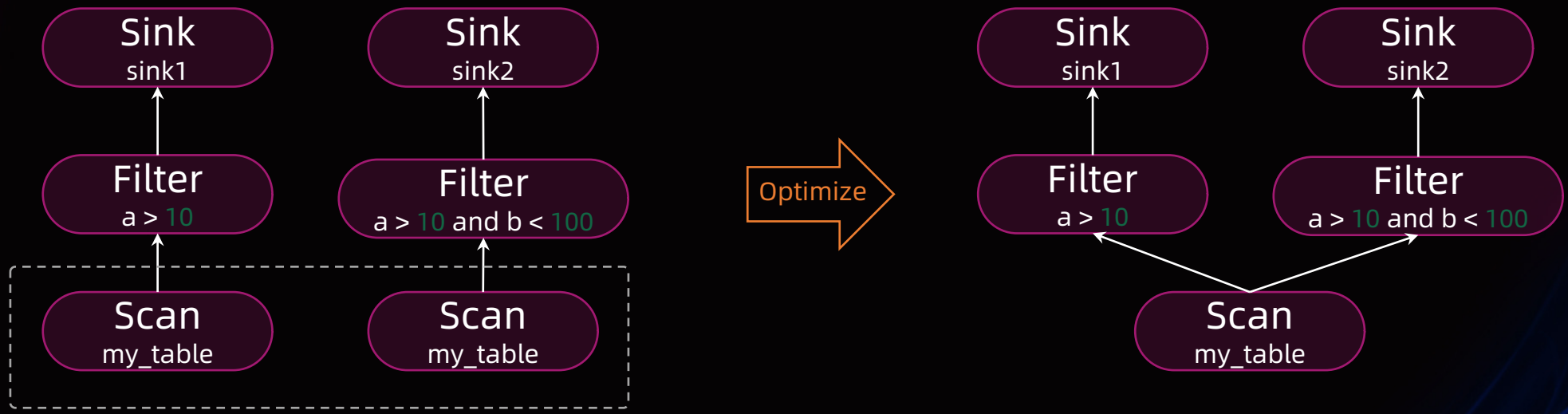
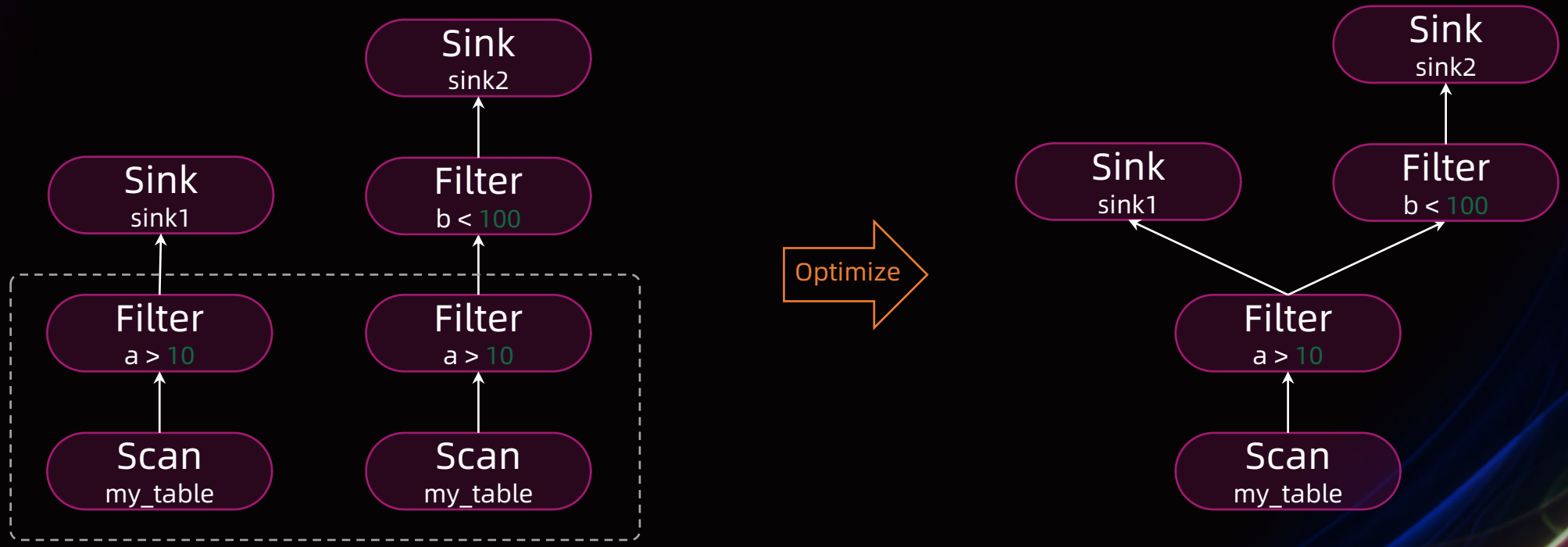


table.optimizer.reuse-sub-plan-enabled： true

Sub-Plan Reuse

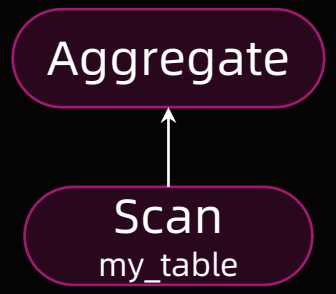
```
create temporary view v1 select * from my_table where a > 10;
insert into sink1 select * from v1;
insert into sink2 select * from v1 where b < 100;
```

Duplicate Computing

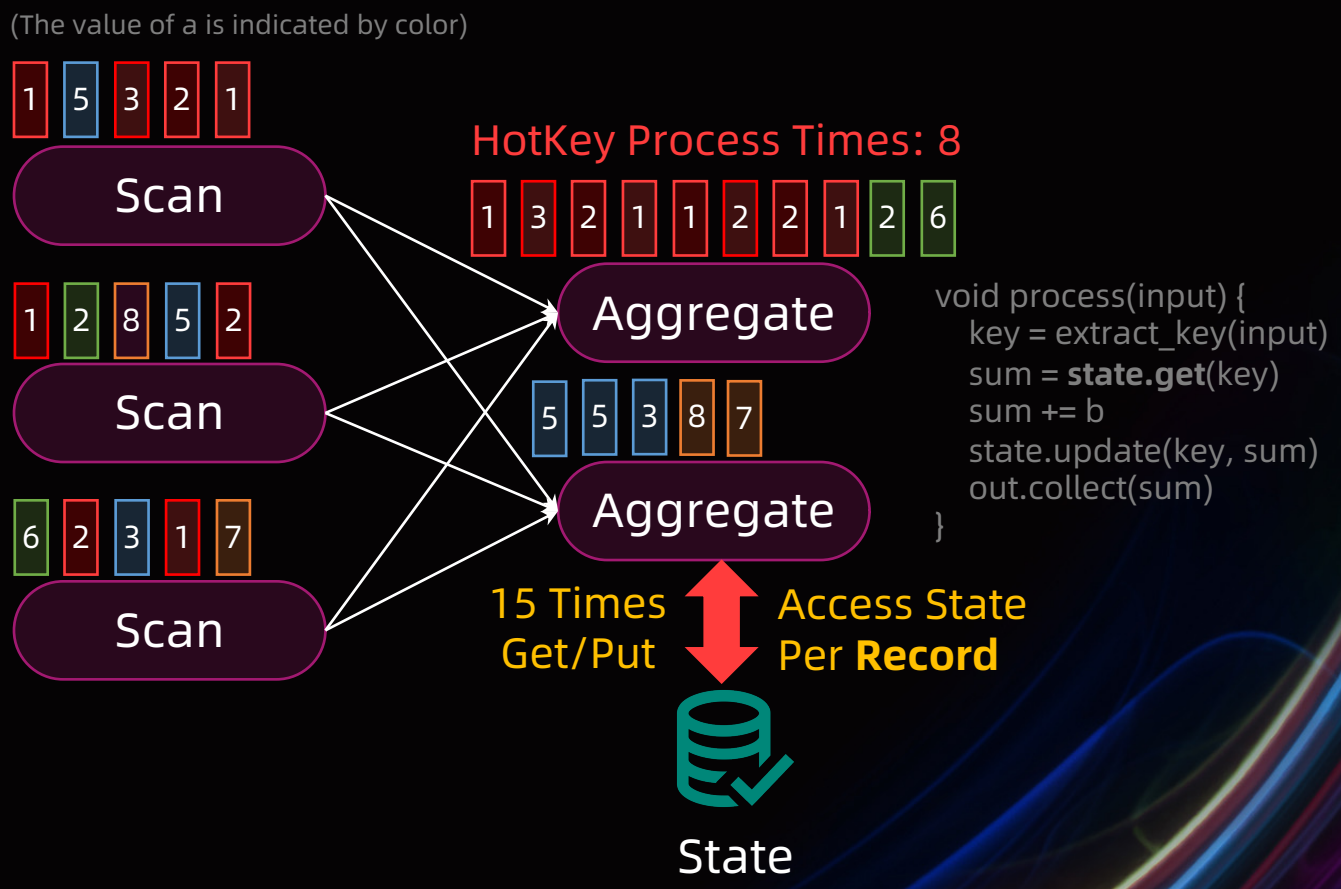


Fast Aggregation

```
select a, sum(b) from my_table group by a;
```



LogicalPlan



Fast Aggregation

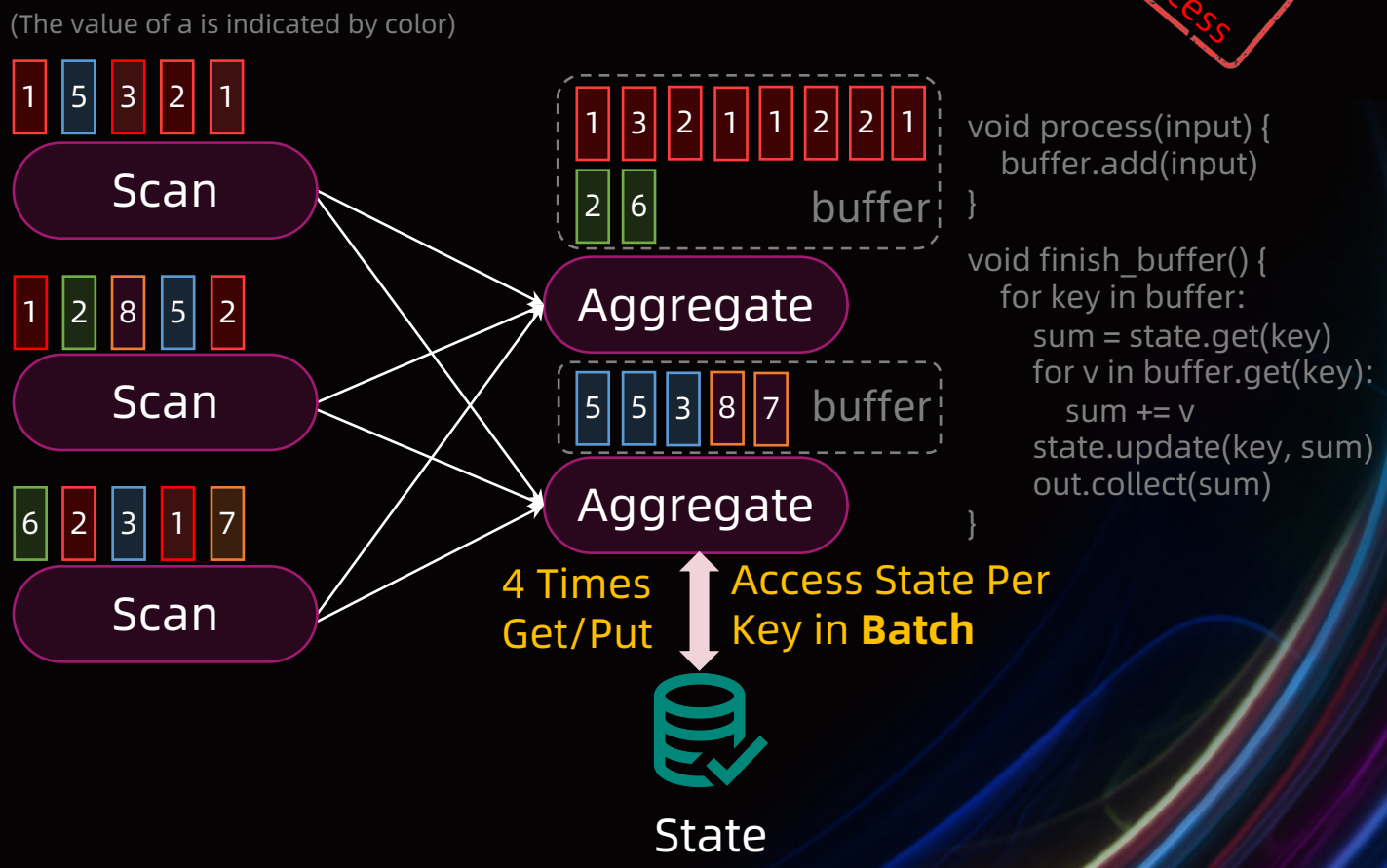
```
select a, sum(b) from my_table group by a;
```

Enable MiniBatch:

- table.exec.mini-batch.enabled: true
- table.exec.mini-batch.allow-latency: 5s

Suitable scenarios:

- Low requirements for job delays (delays in collecting a batch)
- Insufficient state processing capability (Reduce state access)
- Insufficient downstream processing capability (Output less records)



Fast Aggregation

```
select a, sum(b) from my_table group by a;
```

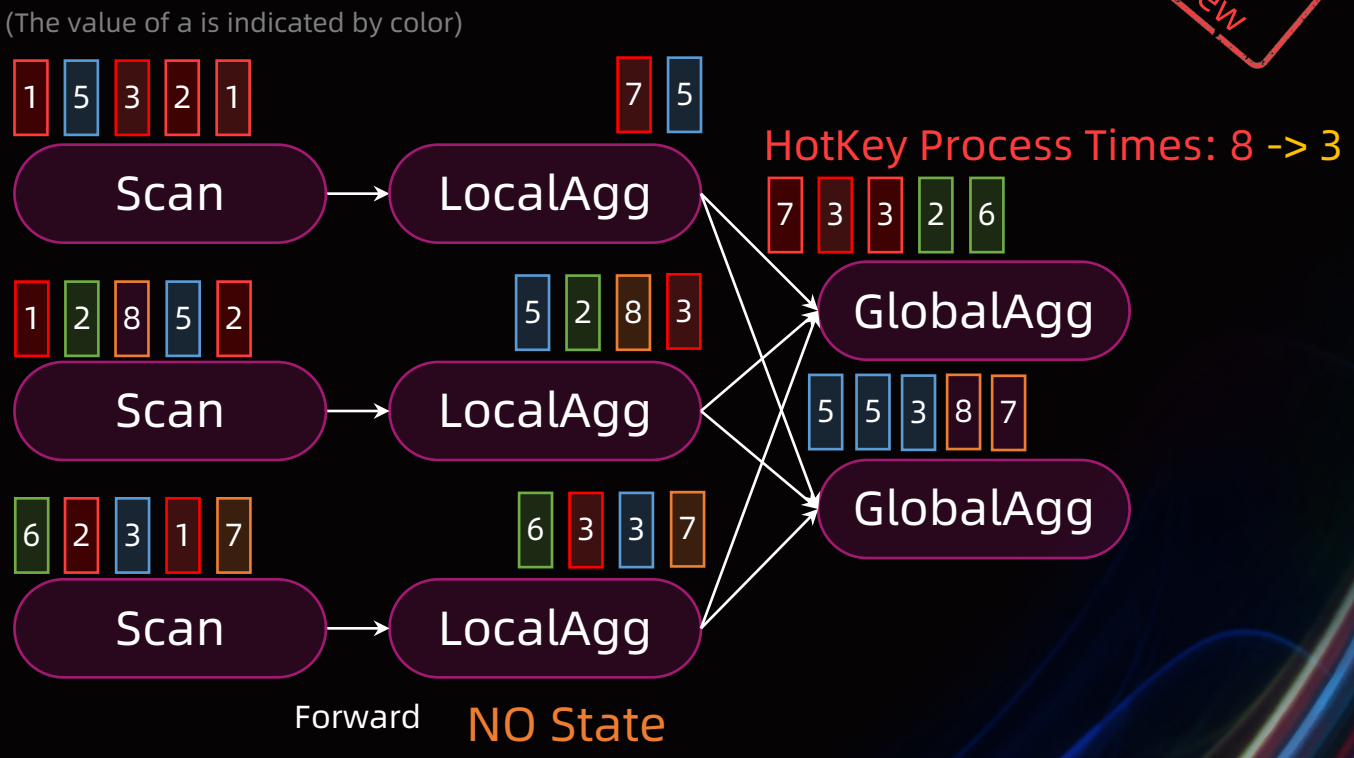
Solve Data Skew

Enable Local/Global:

- table.exec.mini-batch.enabled: true
- table.exec.mini-batch.allow-latency: 5s
- table.optimizer.agg-phase-strategy: TWO_PHASE/AUTO

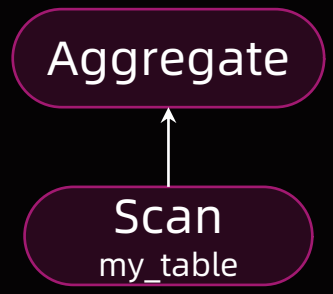
Suitable scenarios:

- All Agg Functions must implement the *merge* method
- Data skew after shuffle

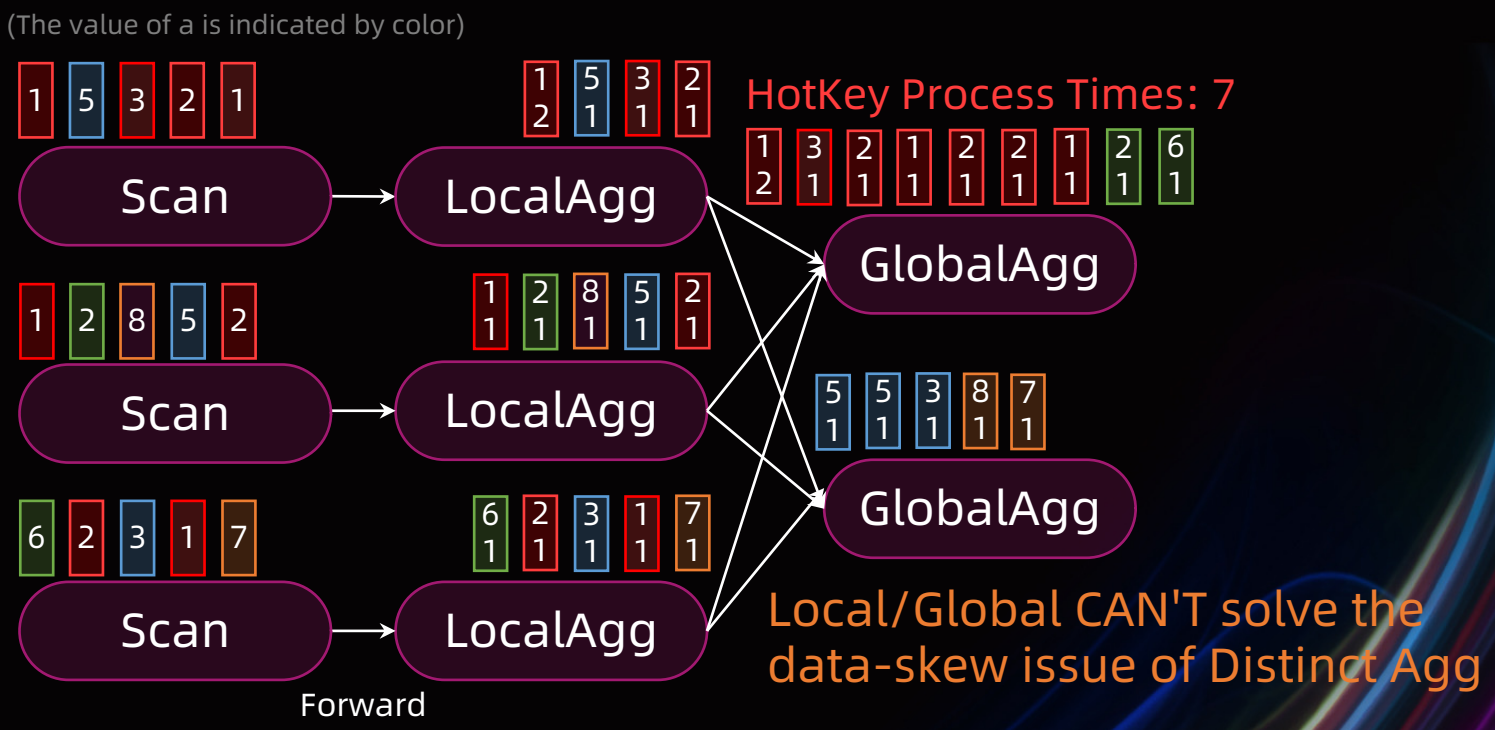


Fast Aggregation

```
select a, count(distinct b) from my_table group by a;
```



LogicalPlan



Fast Aggregation

```
select a, count(distinct b) from my_table group by a;
```

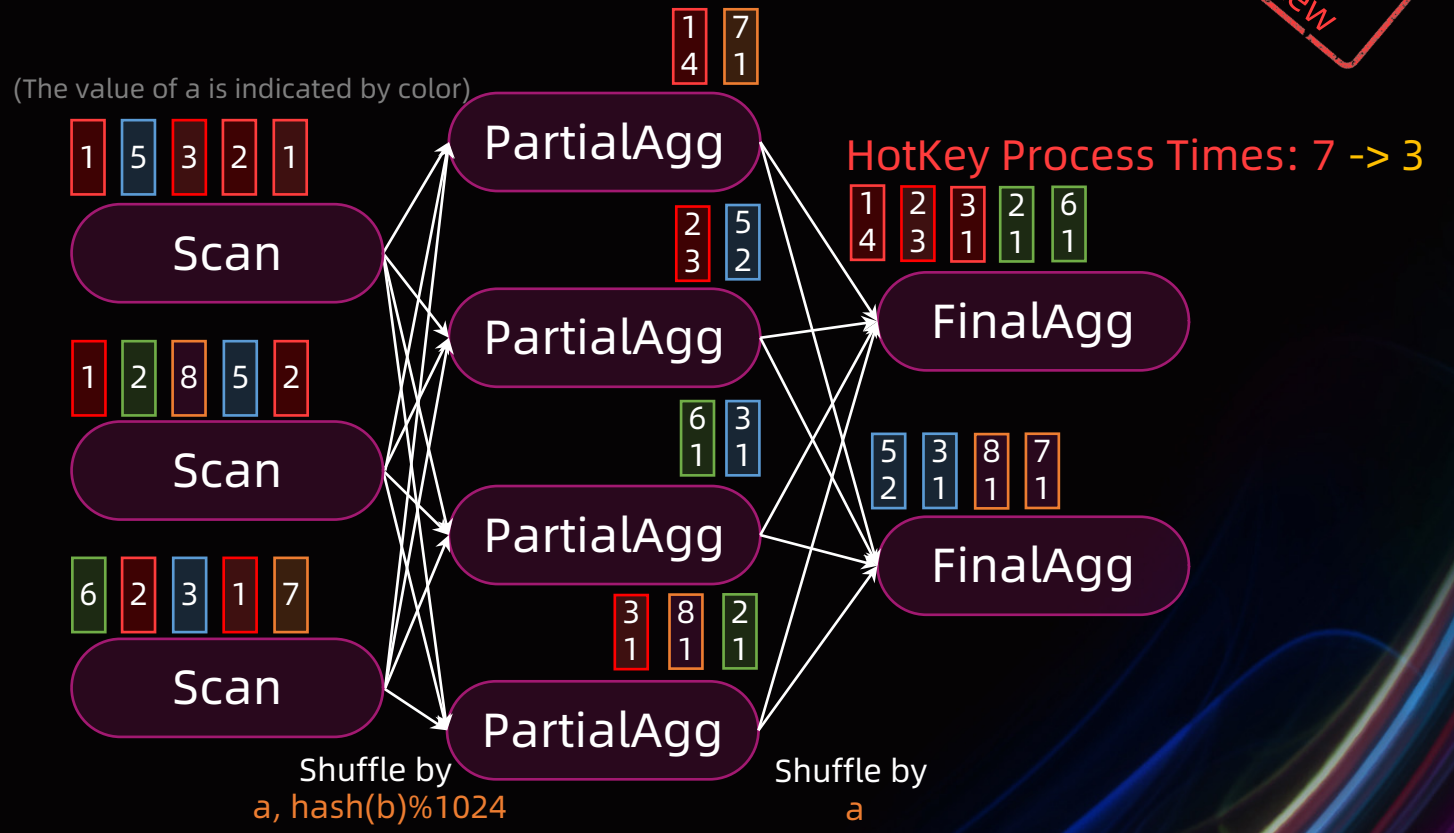
Solve Data Skew

Enable Partial/Final:

- table.optimizer.distinct-agg.split.enabled: true
- table.optimizer.distinct-agg.split.bucket-num: 1024

Suitable scenarios:

- Data skew in distinct function
- Only some built-in UDAFs are supported (e.g. sum, max)
- Large data set is beneficial (additional Shuffle is introduced)
- Note: Partial Agg will add computation and state



Fast Aggregation

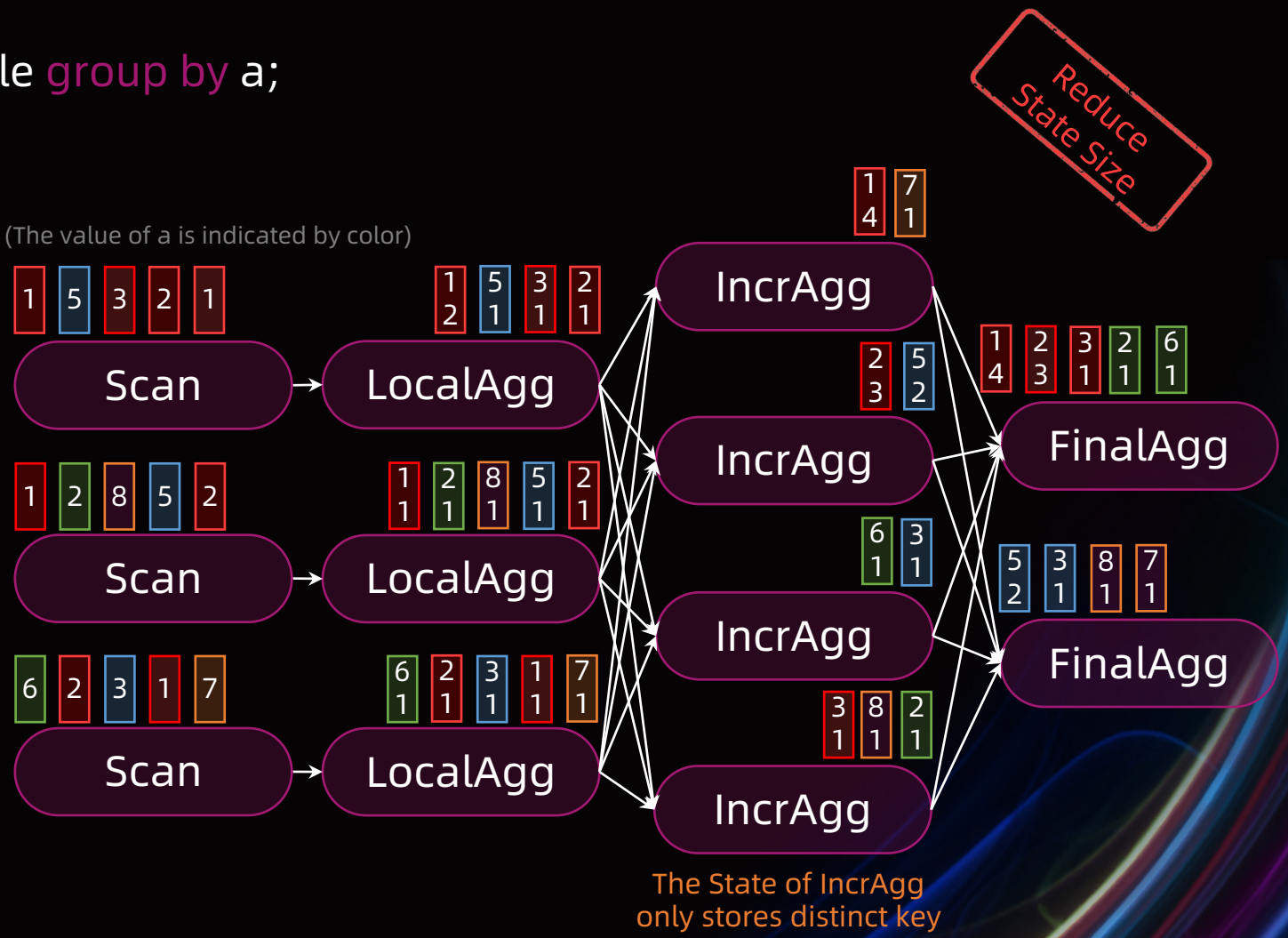
```
select a, count(distinct b) from my_table group by a;
```

Enable Incremental:

- table.exec.mini-batch.enabled: true
- table.exec.mini-batch.allow-latency: 5s
- table.optimizer.agg-phase-strategy: TWO_PHASE/AUTO
- table.optimizer.distinct-agg.split.enabled: true
- table.optimizer.distinct-agg.split.bucket-num: 1024
- table.optimizer.incremental-agg-enabled: true

Suitable scenarios:

- The state size of partial aggregate is too large



Fast Aggregation

Reduce State Size

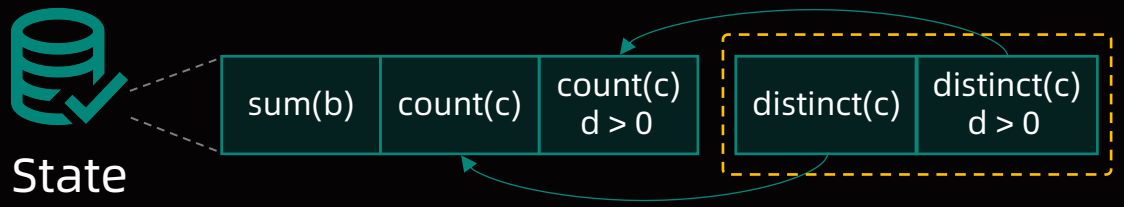
Aggregate Function with CASE WHEN

```
select
  a,
  sum(b),
  count(distinct c) as c1,
  count(distinct
    case when d > 0 then c
    else null end) as c2
from my_table
  group by a;
```

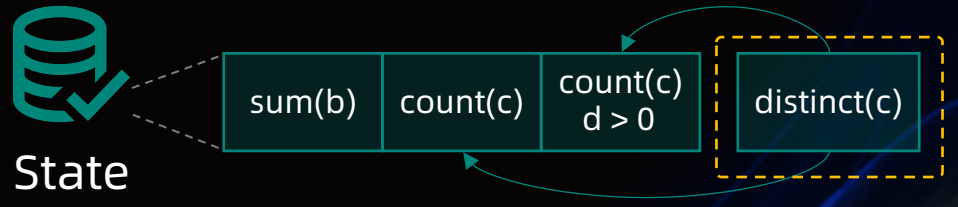
Rewrite

Aggregate Function with FILTER

```
select
  a,
  sum(b),
  count(distinct c) as c1,
  count(distinct c) filter (
    where d > 0) as c2
from my_table
  group by a;
```



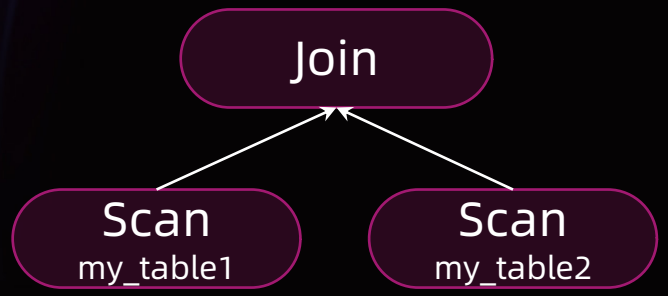
The State corresponding to the same function with same distinct field is stored **independently**



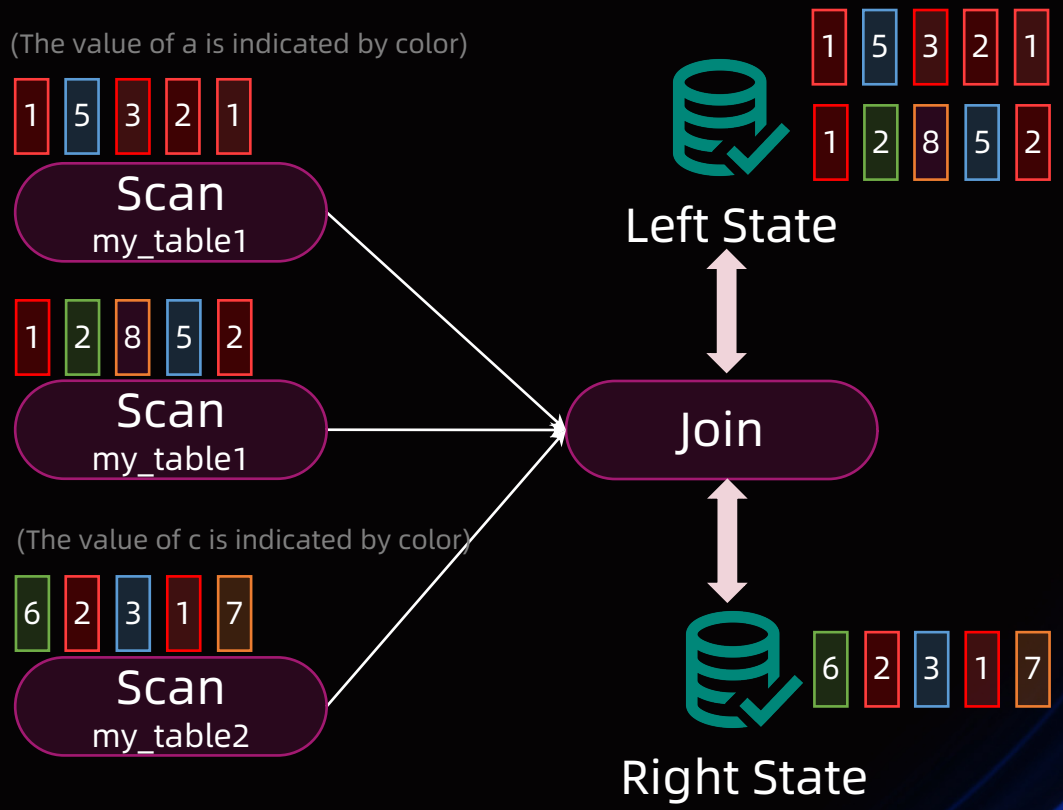
The State corresponding to the same function with same distinct field is stored **shared**

Fast Join

```
select * from my_table1 t1 join my_table2 t2 where t1.a = t2.c;
```



LogicalPlan



The State of the regular Join stores **all the input data**, which cannot handle large data sets

Fast Join

```
select * from my_table1 t1 join my_table2 t2 where t1.a = t2.c;
```

The suggestions to reduce the State of regular Join:

- Join Key contains PK OR Join Input has PK
- Only keep the necessary fields before Join

Join Key => Tuple2<Row, Int>
(The tuple2 is the input row
and its number of associations)

State

Join Key contains PK

Join Key => Map<PK, Tuple2<Row, Int>>
(The Map key is PK, the value is the input row
and its number of associations)

State

Join Input has PK

Join Key => Map<Row, Tuple2<Int, Int>>
(The Map key is the input row, the value is
the pair of the number of occurrences
and the number of associations)

State

No PK

Fast Join

When meeting business requirements, regular Join can be rewritten as follows

Interval Join

The State will only store the data within the Interval, the expired state will be cleared

```
select * from my_table1 t1
join my_table2  t2
on t1.a = t2.c and t1.proctime >
t2.protime - interval '5' second
```

```
select * from (
  select * from table(tumble(table my_table1),
    descriptor(row_time), interval '15' minute
)  t1 join (select ...) t2
on   t1.window_start = t2.window_start
and t1.window_end = t2.window_end
and t1.a = t2.c
```

Lookup Join

```
select * from my_table1 as t1
join my_table2
for system_time as of t1.proctime as t2
on t1.a = t2.c
```

NO State, Good Performance

Regular Join

Window Join

The State will only store the data within the Window, the expired state will be cleared

```
select * from my_table1 as t1
join my_table2
for system_time as of t1.row_time as t2
on t1.a = t2.c
```

Temporal Join

The expired state will be cleared

Fast Lookup Join

```
select * from my_table1 as t1 join my_table2
for system_time as of t1.proctime as t2 on t1.a = t2.c;
```

Improve Throughput

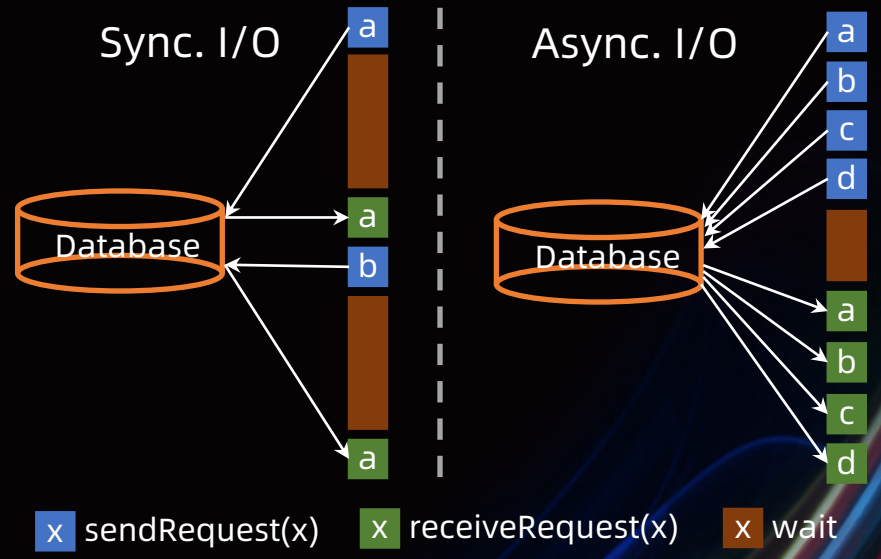
Sync vs Async

- Sync:

```
select /*+ LOOKUP('table'='my_table2', 'async'='false') */ *
from my_table1 as t1 join my_table2
for system_time as of t1.proctime as t2 on t1.a = t2.c;
```

- Async (Improve throughput):

```
select /*+ LOOKUP('table'='my_table2', 'async'='true') */ *
from my_table1 as t1 join my_table2
for system_time as of t1.proctime as t2 on t1.a = t2.c;
```



Fast Lookup Join

```
select * from my_table1 as t1 join my_table2
for system_time as of t1.proctime as t2 on t1.a = t2.c;
```

Improve Throughput

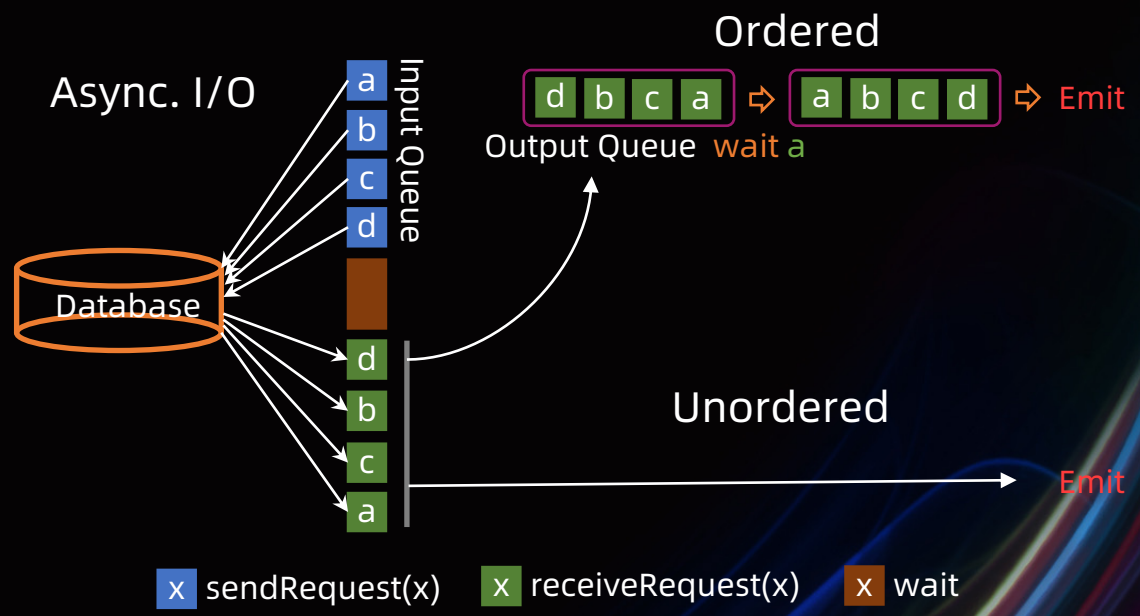
Async Output Mode:

- Ordered:

```
/*+ LOOKUP('table'='my_table2', 'async'='true',
    'output-mode'='ordered', 'capacity'='100',
    'timeout'='180s') */
```

- Unordered (Improve throughput):

```
 /*+ LOOKUP('table'='my_table2', 'async'='true',
    'output-mode'='allow_unordered',
    'capacity'='100', 'timeout'='180s') */
```



Job Level Config:

```
table.exec.async-lookup.output-mode: ALL_UNORDERED
table.exec.async-lookup.buffer-capacity: 100
table.exec.async-lookup.timeout: 180s
```

Fast Lookup Join


```
select * from my_table1 as t1 join my_table2
for system_time as of t1.proctime as t2 on t1.a = t2.c;
```

Cache (Memory Lookup):

- Full Caching (Small data set):
  'lookup.cache': 'FULL'
  'lookup.full-cache.reload-strategy': 'PERIODIC'
  'lookup.full-cache.periodic-reload.interval': '1h'
  'lookup.full-cache.periodic-reload.schedule-mode': 'FIXED_DELAY'

- Partial Caching (Large data set):
  'lookup.cache': 'PARTIAL'
  'lookup.partial-cache.max-rows': 10000
  'lookup.partial-cache.expire-after-access': '1h'
  'lookup.partial-cache.expire-after-write': '1h'
  'lookup.partial-cache.cache-missing-key': ‘true’

- No Caching (Disable cache):
  'lookup.cache': 'NONE’

Fast Deduplication

Find the FIRST row per key

```
select
  a,
  first_value(b),
  first_value(c)
from my_table
  group by a;
```

Find the LAST row per key

```
select
  a,
  last_value(b),
  last_value(c)
from my_table
  group by a;
```

BAD Practices

- Large State
- Incomplete semantics (can't handle *null* values)

Fast Deduplication

Find the FIRST row per key

```
select  a, b, c from (
  select a, b, c,
    row_number() over (
      partition by a
      over by time_attr asc) as rn
    from my_table)
where rn = 1;
```

The State will only store Keys

Find the LAST row per key

```
select  a, b, c from (
  select a, b, c,
    row_number() over (
      partition by a
      over by time_attr desc) as rn
    from my_table)
where rn = 1;
```

The State will only store the last row of each Key
(NOT supported when upstream is changelog
and time attribute is row_time)

Fast TopN

TopN (Rank) implementations
(Performance decreases sequentially)

- AppendRank
  - Input produces insert-only changes
  - (The State stores the records in TopN per key)
- UpdateFastRank
  - Input produces update changes
  - Upsert key contains partition key
  - Order-by fields are monotonic, and the monotonic direction is opposite to the order-by direction
  - (The State stores a Map, its key is order key and its value is the record and the order number)
- RetractRank
  - NO required for input
  - (The State stores all input data)


```
select  a, b, c from (
 select a, b, c,
  row_number() over (
   -- The upsert key from upstream contains partition key
   partition by a, b
   -- c is monotonically increasing, while the order
   -- direction is monotonically decreasing
   order by c desc) as rn
 from (
  select a, b,
   -- Declare the argument of sum to be a positive number,
   -- So that the result of sum is monotonically increasing
   sum(c) filter (where c >= 0) as c
   from my_table
   -- Produce upsert stream
   group by a, b))
where rn < 10;
```

Fast TopN

TopN optimization methods

- DO NOT output row_number field
  - Greatly reduces the amount of data in the result table
  - (the data can be sorted in front-end if needed)
- Increase Cache (LRU memory) size
  - table.exec.rank.topn-cache-size: 10000
  - cache_hit = cache_size * parallelism / top_n_num / partition_key_num
  - (NOTE: The TM memory needs to be increased accordingly)
- Partition fields are best related to time
  - Otherwise, state TTL will leads to wrong result

Efficient User Defined Connector

Implement the following interfaces to improve execution efficiency:

- SupportsFilterPushDown
  - Reduce Scan I/O
- SupportsProjectionPushDown
  - Reduce invalid field reads
- SupportsPartitionPushDown
  - Reduce the invalid Partitions (static optimization)
- SupportsDynamicFiltering
  - Reduce the invalid Partitions (dynamic optimization)
- SupportsLimitPushDown
  - Reduce Scan I/O
- SupportsAggregatePushDown
  - Reduce Scan I/O, output less data
- SupportsStatisticReport
  - Report statistics, the optimizer can generate a better execution plan

Use Hints Well

- Table Hints：Change Table Options
  - e.g. Change the cache strategy of Lookup Table
    - /*+ OPTIONS('lookup.cache'='FULL') */
- Query Hints：
  - LOOKUP: Change Lookup Join Strategy
    - /*+ LOOKUP('table'='my_table2', 'async'='true') */
  - BROADCAST: Suggest the Optimizer chooses Broadcast Hash Join （batch only）
    - /*+ BROADCAST(t1)*/
  - SHUFFLE_HASH: Suggest the Optimizer chooses Shuffle Hash Join （batch only）
    - /*+ SHUFFLE_HASH(t1)*/
  - SHUFFLE_MERGE: Suggest the Optimizer chooses Sort Merge Join （batch only）
    - /*+ SHUFFLE_MERGE(t1)*/
  - NEST_LOOP: Suggest the Optimizer chooses Nested Loop Join （batch only）
    - /*+ NEST_LOOP(t1)*/

03 Future Works

Future Works

Deeper Optimization

Support more advanced optimizations

Richer Optimization

Support more scenarios

Smarter Optimization

Support dynamic optimization

FLINK FORWARD # ASIA 2022 ONLINE | 实时即未来

THANK YOU

谢　谢　观　看