2023

如何打造一款极速数据库

康凯森

2023-11-22

康凯森

- StarRocks Query Team Leader
- StarRocks PMC & Committer
- Apache Kylin PMC & Committer (Inactive)
- Apache Doris PMC & Committer (Retired)

如何打造一款极速数据库

01 核心思路

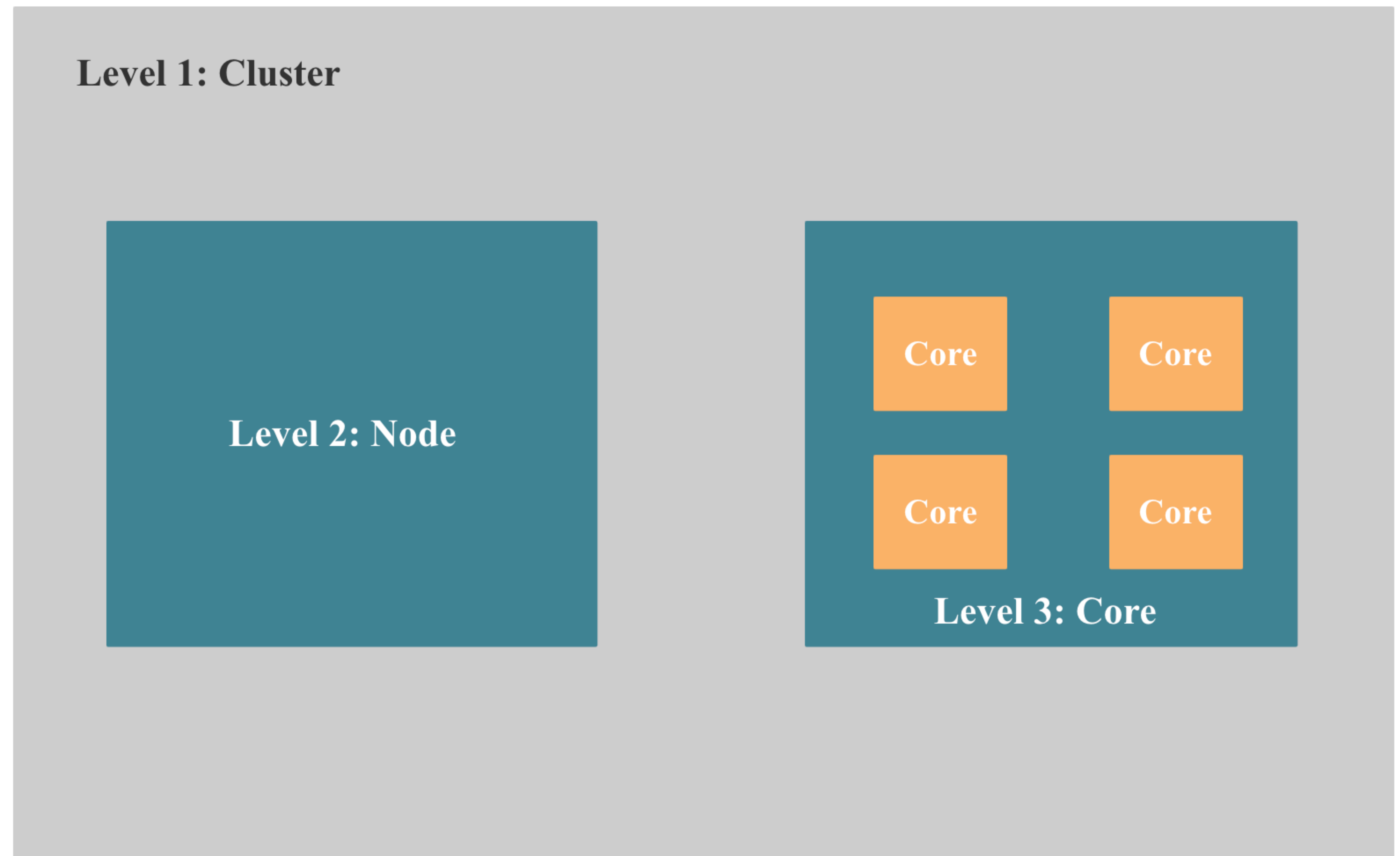
2 关键点

3 总结

4 讨论

如何打造极速数据库 1：架构视角

- Level 1: 弹性伸缩
- Level 2: 多机扩展性
- Level 3: 多核扩展性
- Level 4: 单核性能



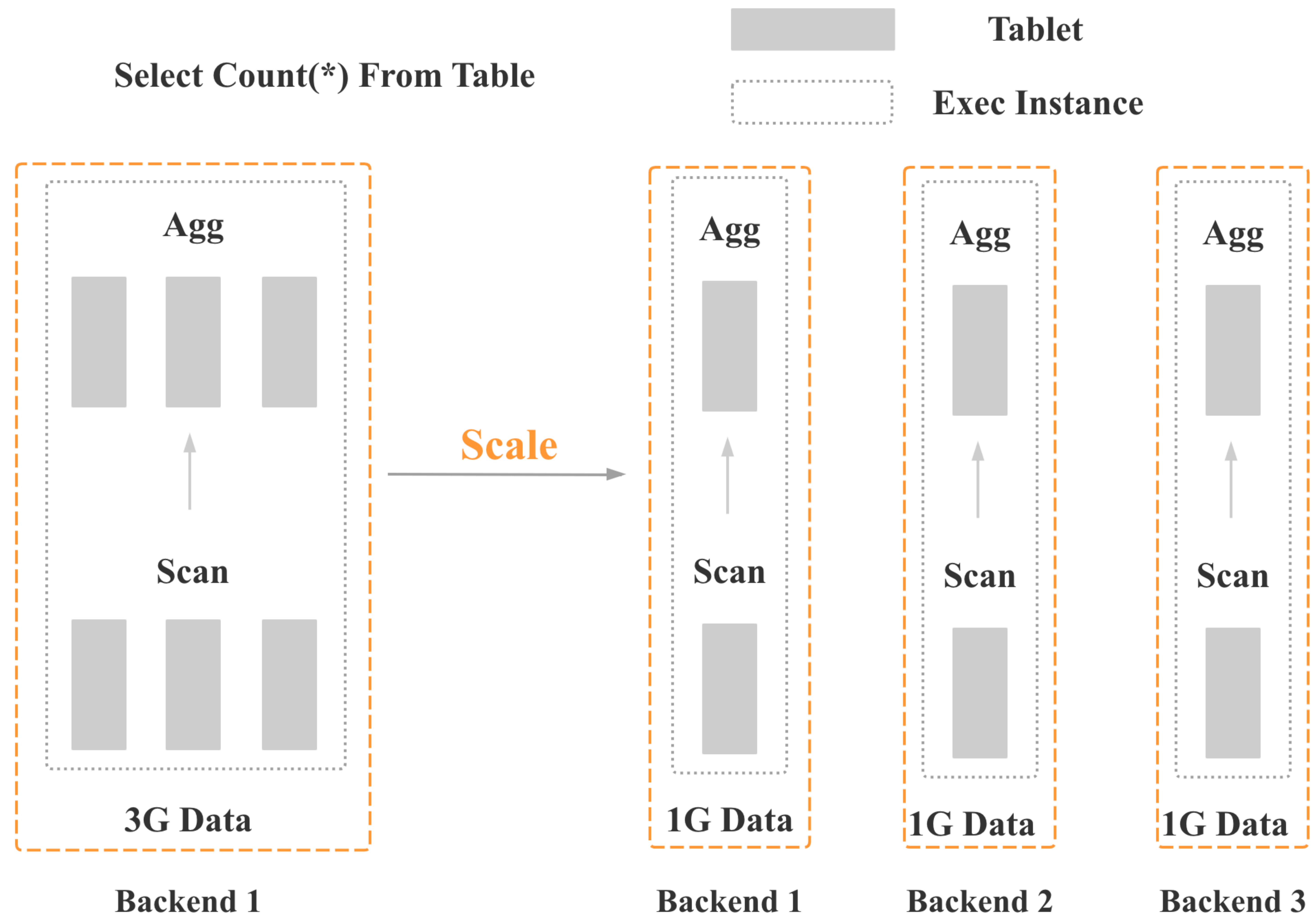
极速数据库：弹性伸缩

Number of Workers	Cost Per Hour	Length of Workload (hours)	Cost of Workload
1	$1	2	$2
2	$2	1	$2
4	$4	0.5	$2
8	$8	0.25	$2

相同的成本，数倍的性能体验

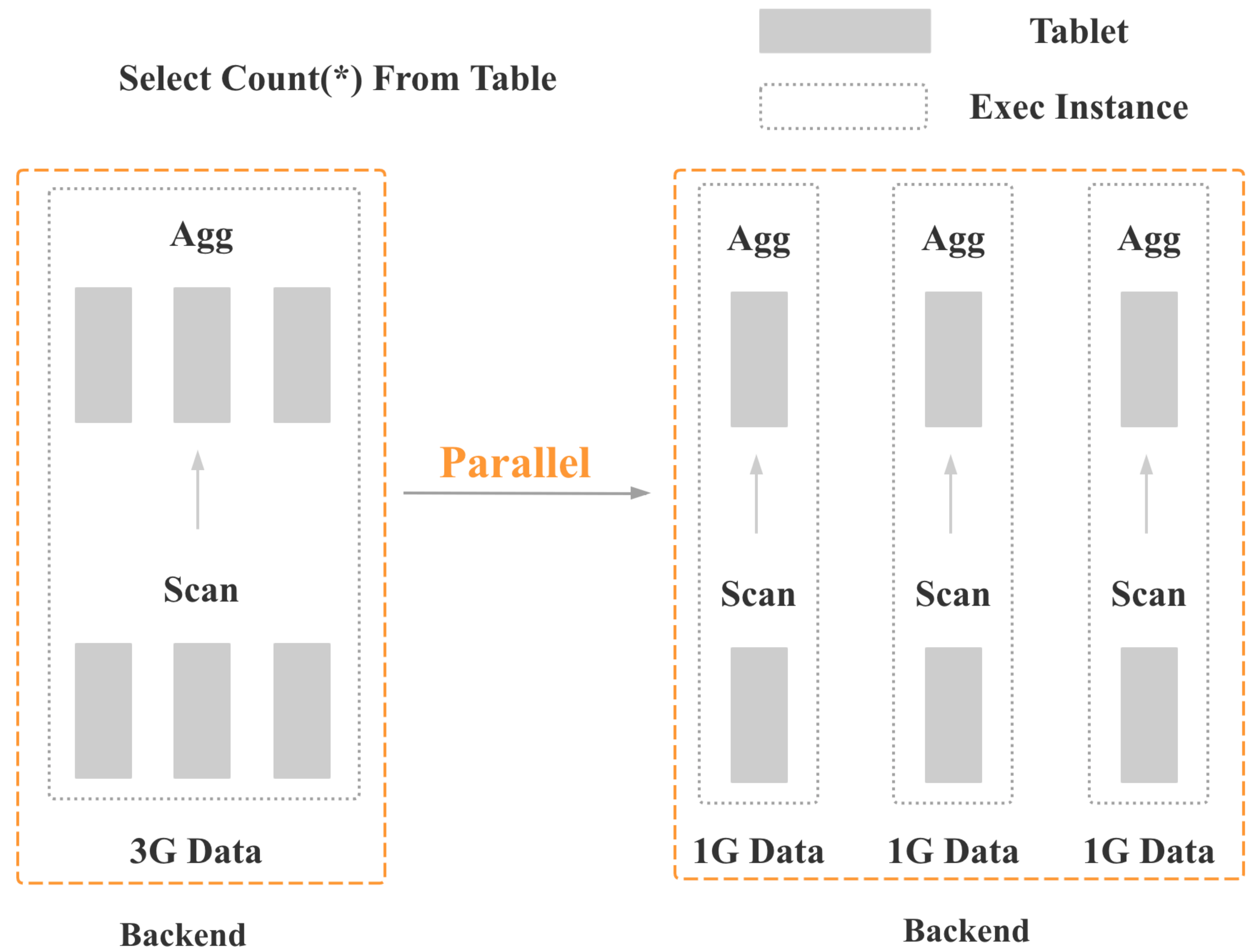
极速数据库：多机优化

- 执行没有单点
- 没有数据倾斜
- 分布式调度开销



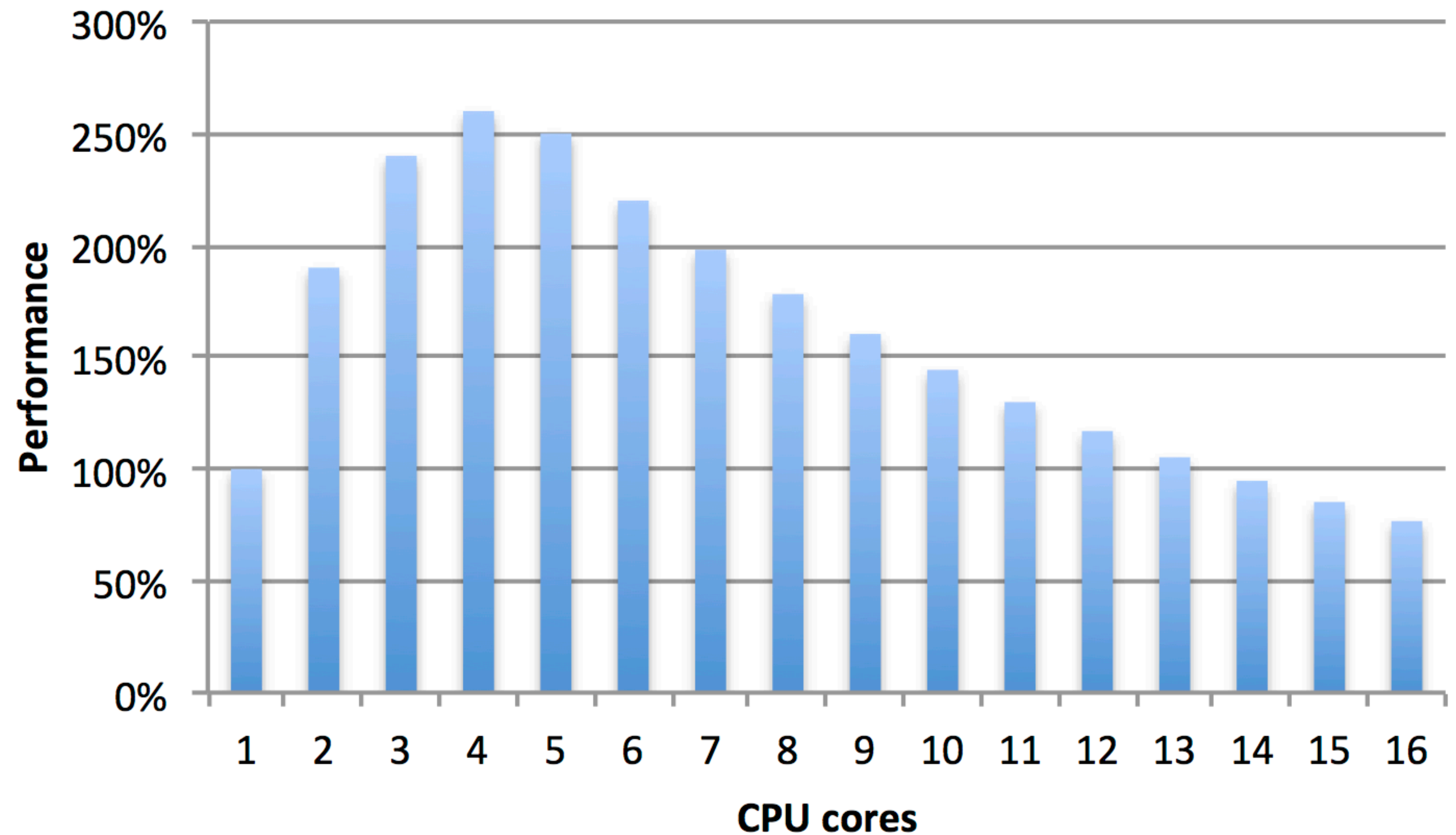
极速数据库：多核优化

- 串行执行要尽可能少
- 没有数据倾斜
- 多核调度开销
- Cache Miss
- NUMA



极速数据库：多核优化

- 串行执行要尽可能少
- 没有数据倾斜
- 多核调度开销
- Cache Miss
- NUMA

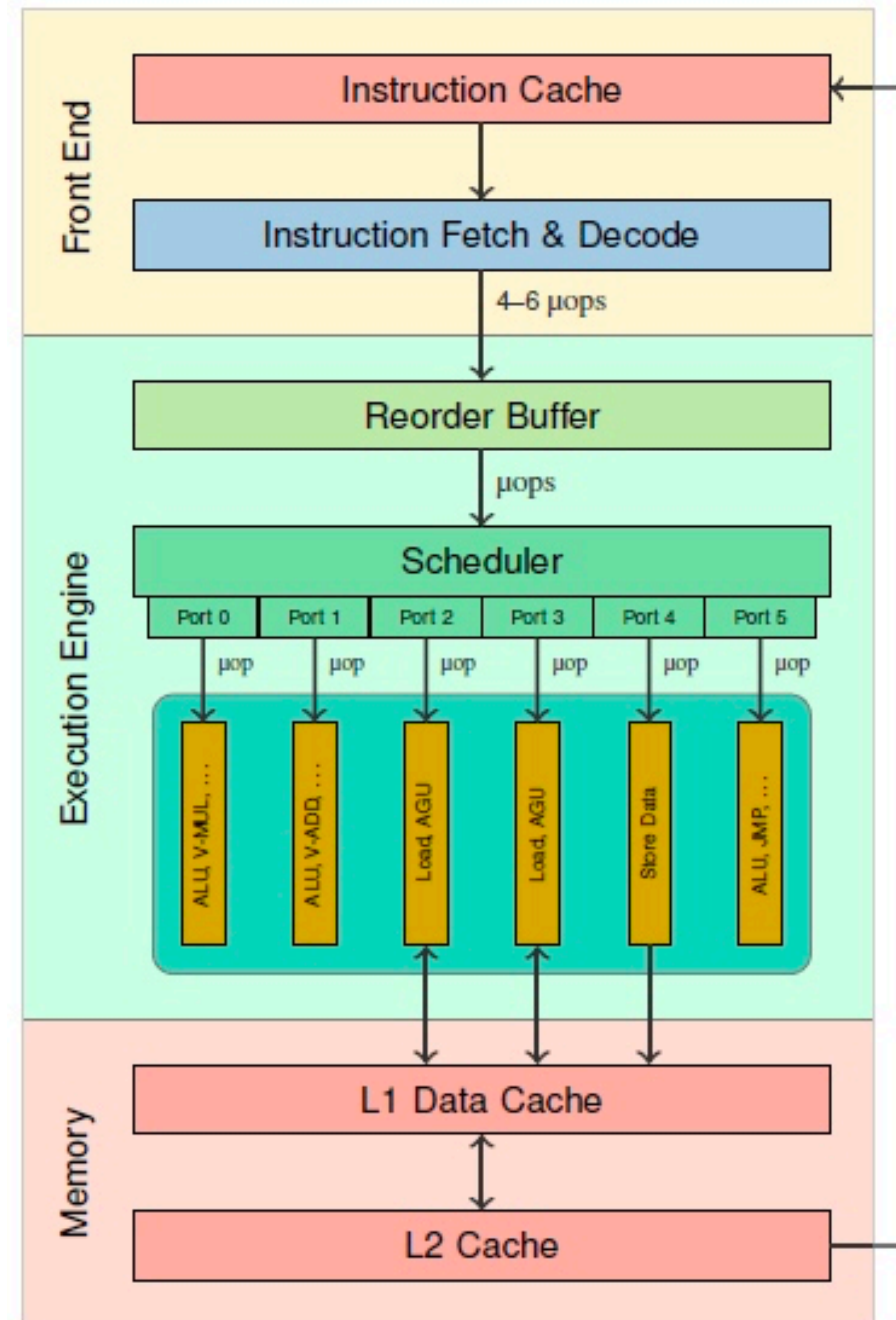


极速数据库：单核优化

CPU Time = Instruction Number * CPI * Clock Cycle Time

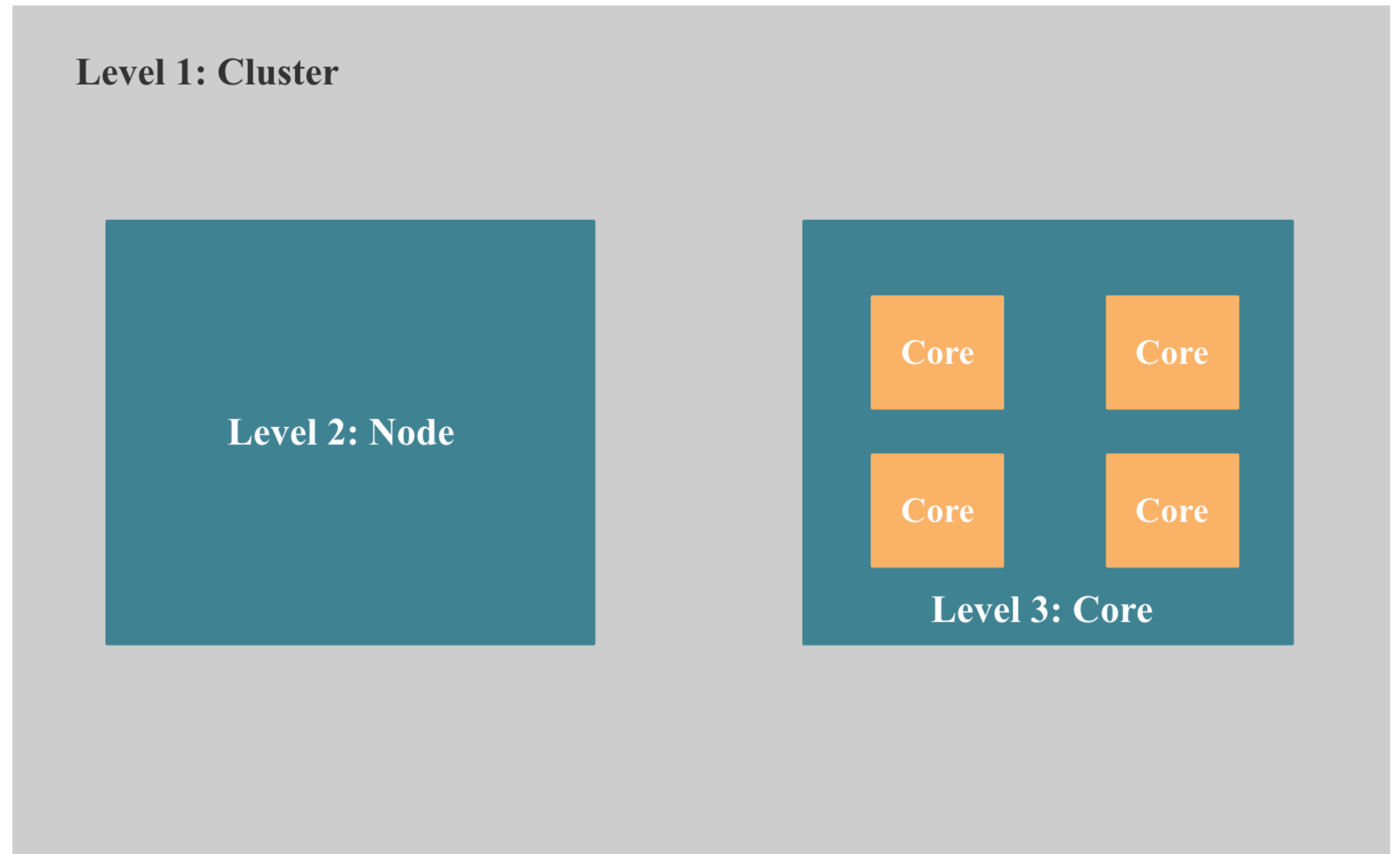
极速数据库：单核优化

- 向量化
- 算法和数据结构
- CPU Cache
- 分支预测

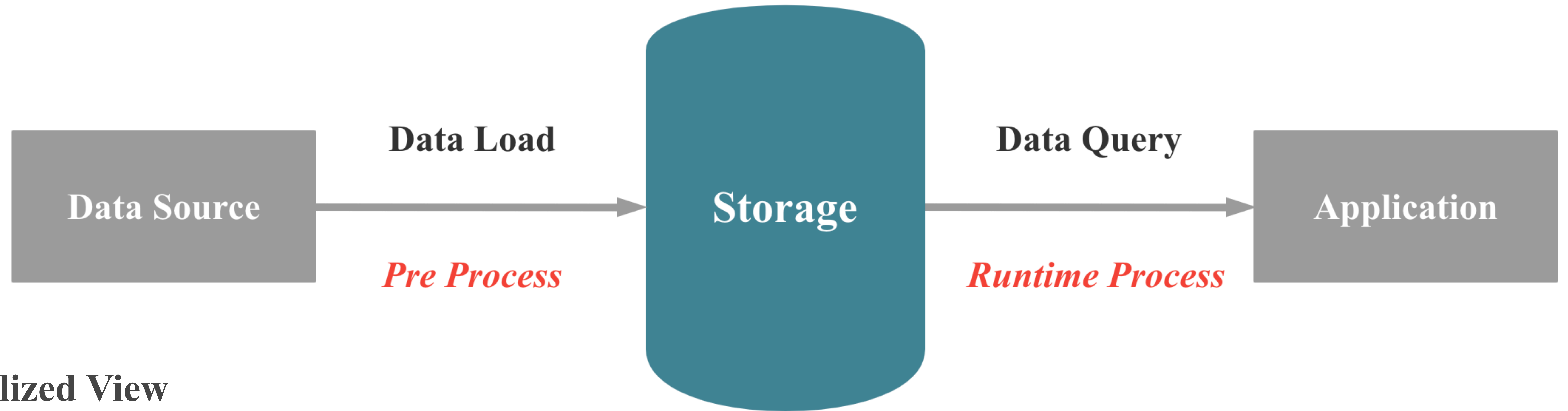


如何打造极速数据库 1：架构视角

- 存算分离&弹性伸缩
- MPP 多机并行执行
- Pipeline 多核并行执行
- 向量化执行 (单核)



如何打造极速数据库 2：Pre Process VS Runtime Process



- Materialized View
- Aggregate Data When Load
- Index
- Cache

The more pre process, The less runtime process

如何打造极速数据库 2：物化视图

Base Table

Year	City	Cost(sum)
2016	beijing	10
2016	shanghai	30
2017	beijing	20
2017	shnaghai	40

MV (Year, Cost)

Year	Cost(sum)
2016	40
2017	60

← **select sum(cost) from table where year = 2016**

MV (City, Cost)

City	Cost(sum)
beijing	30
shanghai	70

← **select sum(cost) from table where city = shanghai**

- 查询透明加速
- 空间换时间

如何打造极速数据库 2：聚合表

Date	PublisherId	Country	Clicks	Cost
2013/12/31	100	US	10	32
2014/01/01	100	US	205	103
2014/01/01	200	UK	100	50

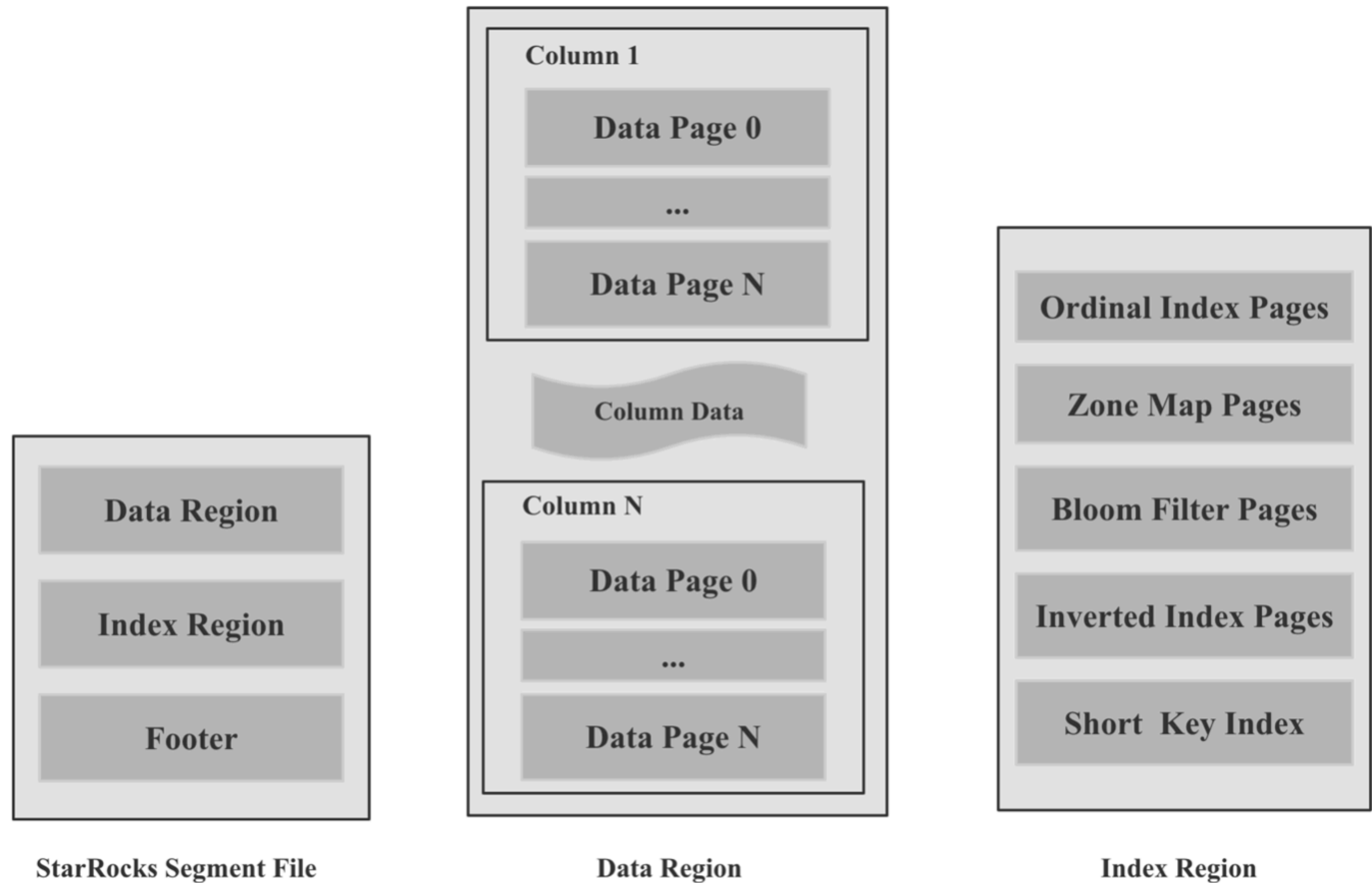
(a) Mesa table A

- Key列全局有序
- 每个Value列有关联的聚合函数(Sum, Max, Min, HLL, Bitmap, Replace)
- 相同Key的Value进行聚合
- 基于Replace函数可以实现主键更新

导入时聚合，极大减少查询时处理的数据量

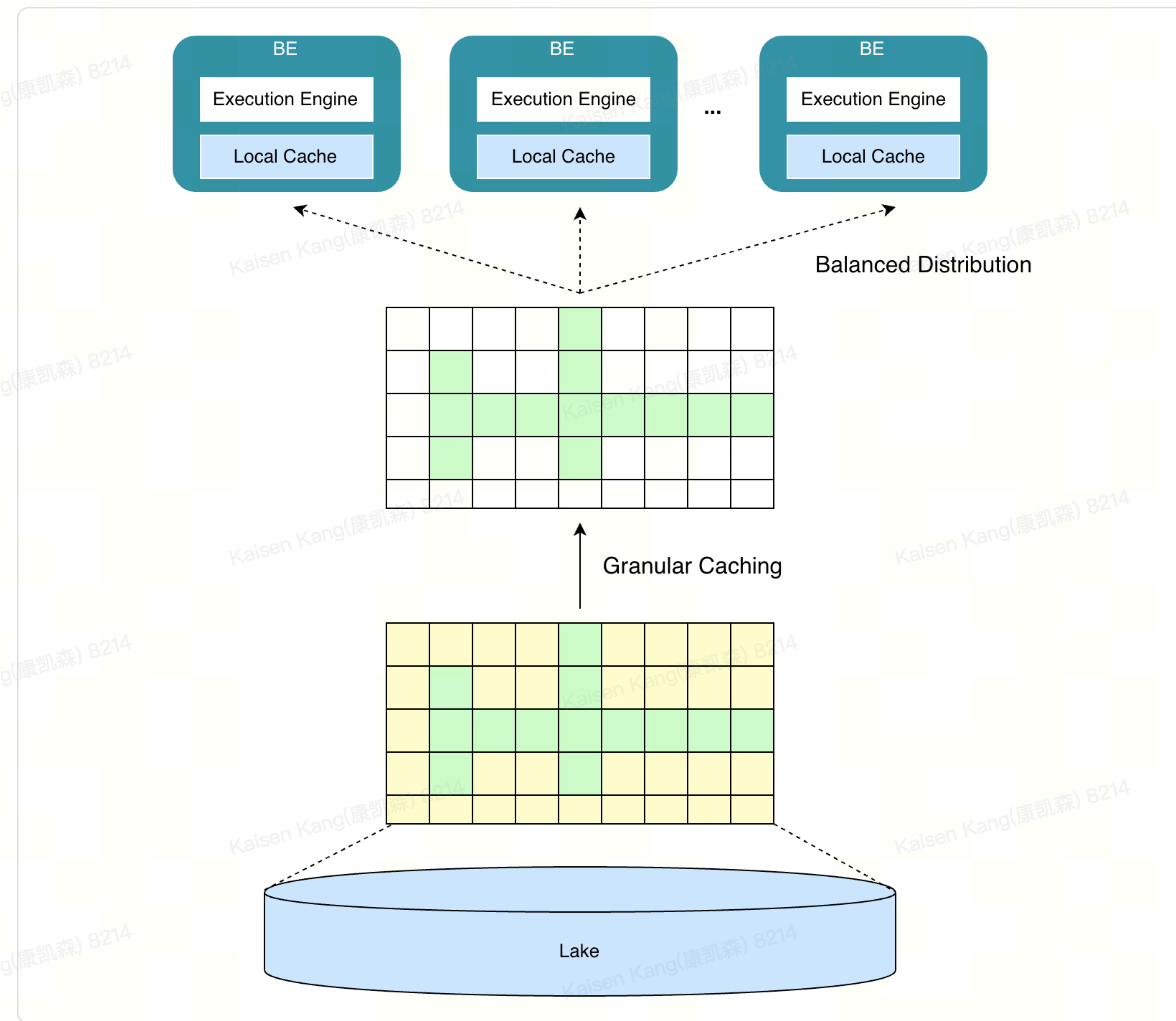
如何打造极速数据库 2：索引

- 减少 Scan 数据量
- 空间换时间
- 如何选择最佳索引类型
- 如何减少随机 IO 次数

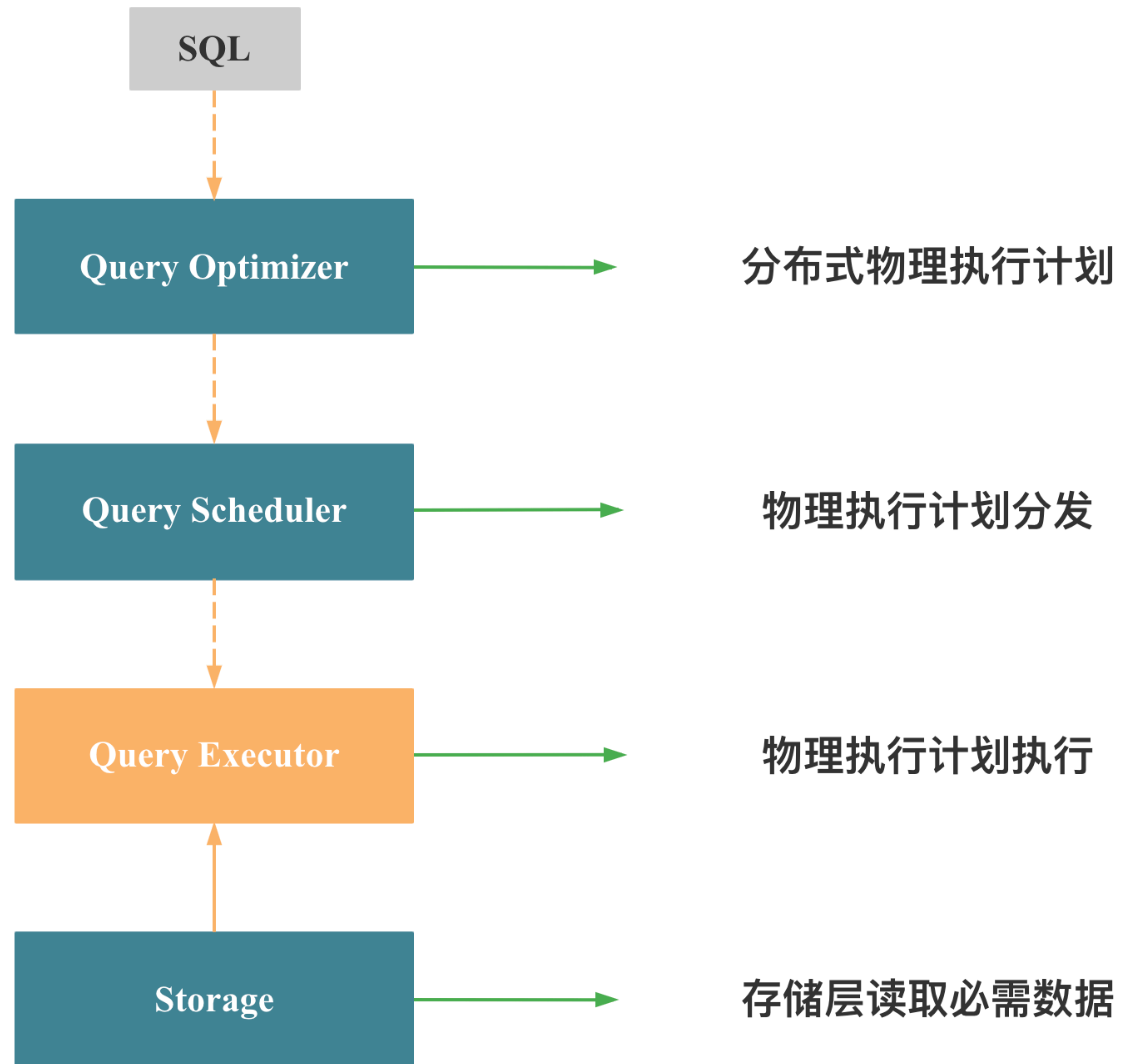


如何打造极速数据库 2：Cache

- Local Disk Cache
- Memory Page Cache
- Query Cache
- Query Result Cache
- MetaData Cache
- Statistics Cache

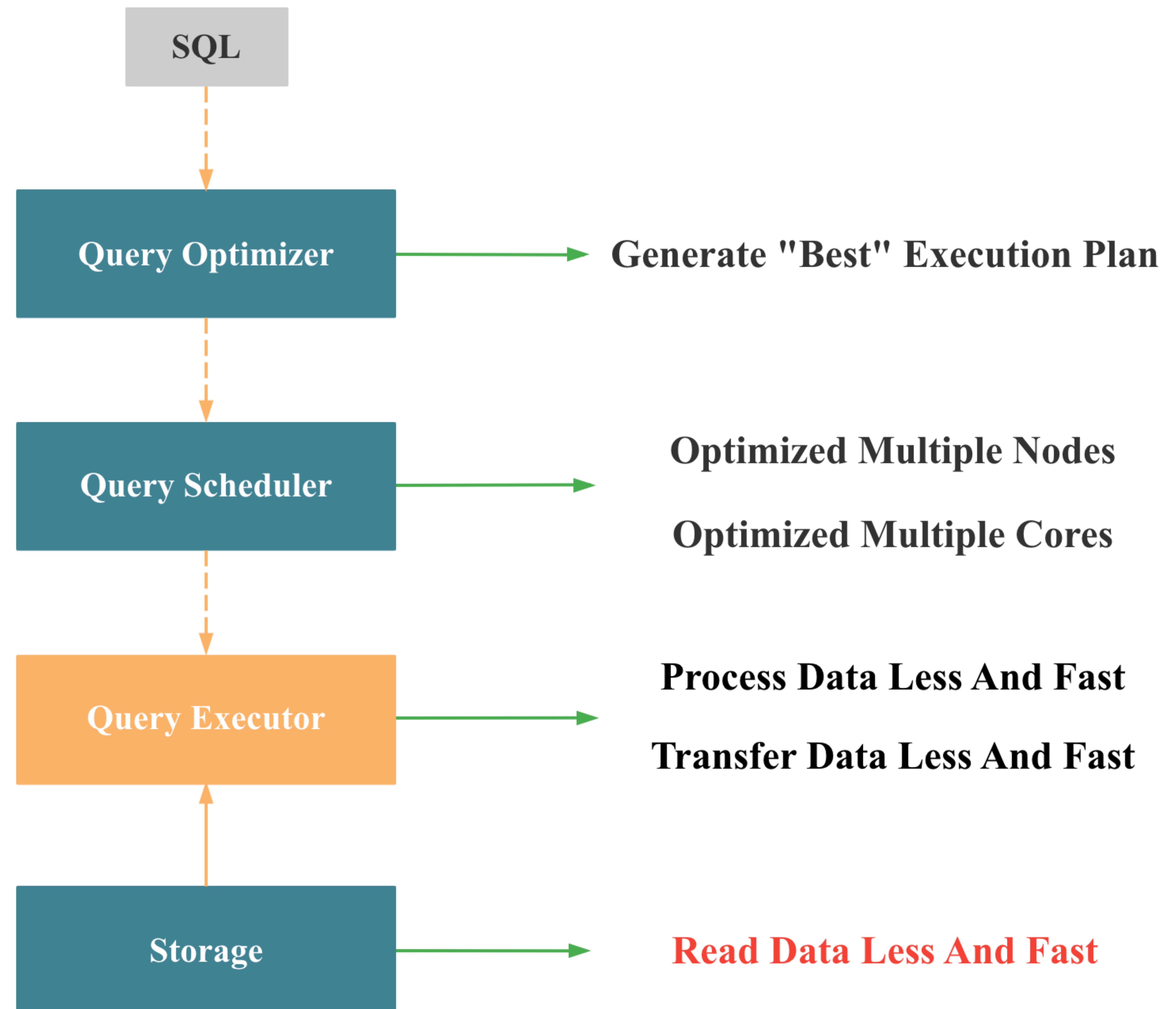


如何打造极速数据库 3：查询数据流 1



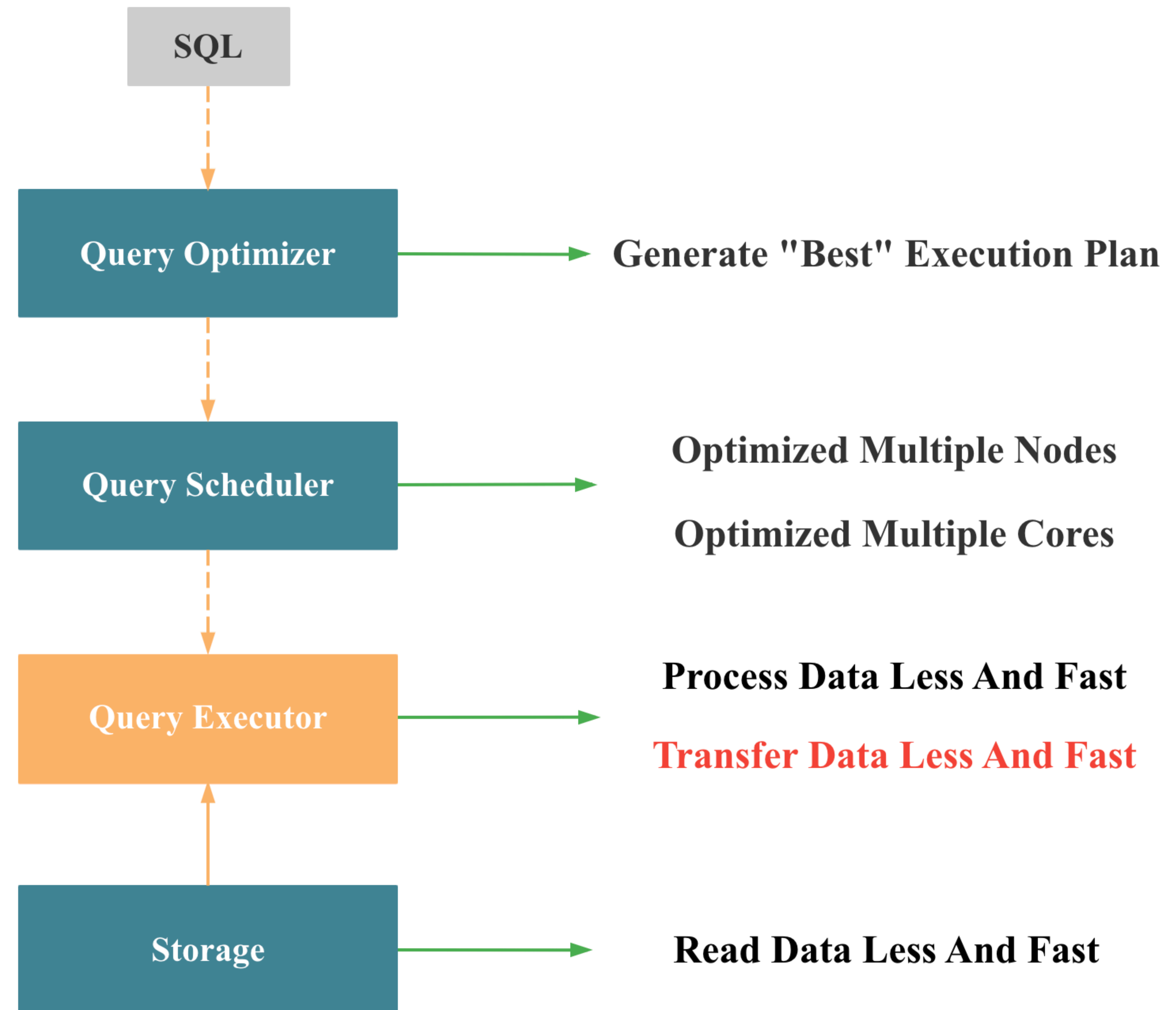
如何打造极速数据库 3：查询数据流 2

- Partition And Bucket Prune
- Read Necessary Column
- Read Compressed Data
- Skip Data By Index
- Skip Data By Metadata
- Late Materialization
- Operations On Encoded Data
- Vectorized Process



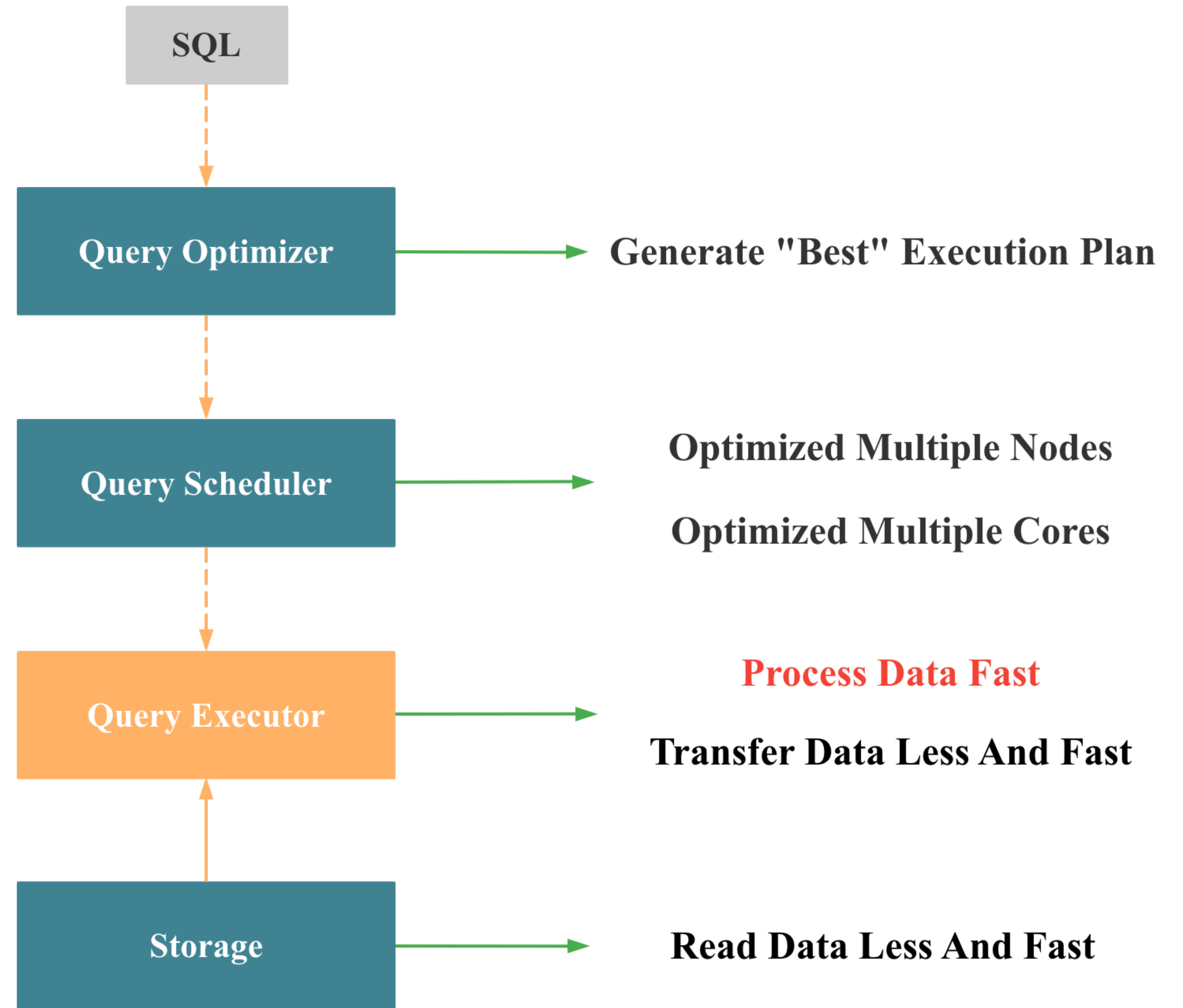
如何打造极速数据库 3：查询数据流 3

- Shuffle By Column
- Compress Data By Column
- Global Runtime Filter
- Operations On Encoded Data
- Colocate Join
- Replication Join
- Bucket Shuffle Join



如何打造极速数据库 3：查询数据流 4

- Data Structure and Algorithms
- Vectorization
- SIMD
- Adaptive Strategy
- Cache Optimization
- C++ Low Level Optimization



如何打造极速数据库 4: 资源视角

- Read Data Less And Fast (IO)
- Transfer Data Less And Fast (Network)
- Process Data Less And Fast (CPU & Memory)

善于利用各种 Linux Tools 诊断性能瓶颈

如何打造极速数据库 5: Adaptive & AI Tuning

- 统计信息不准
- 优化器 Cost 估计不准
- Continuous Query

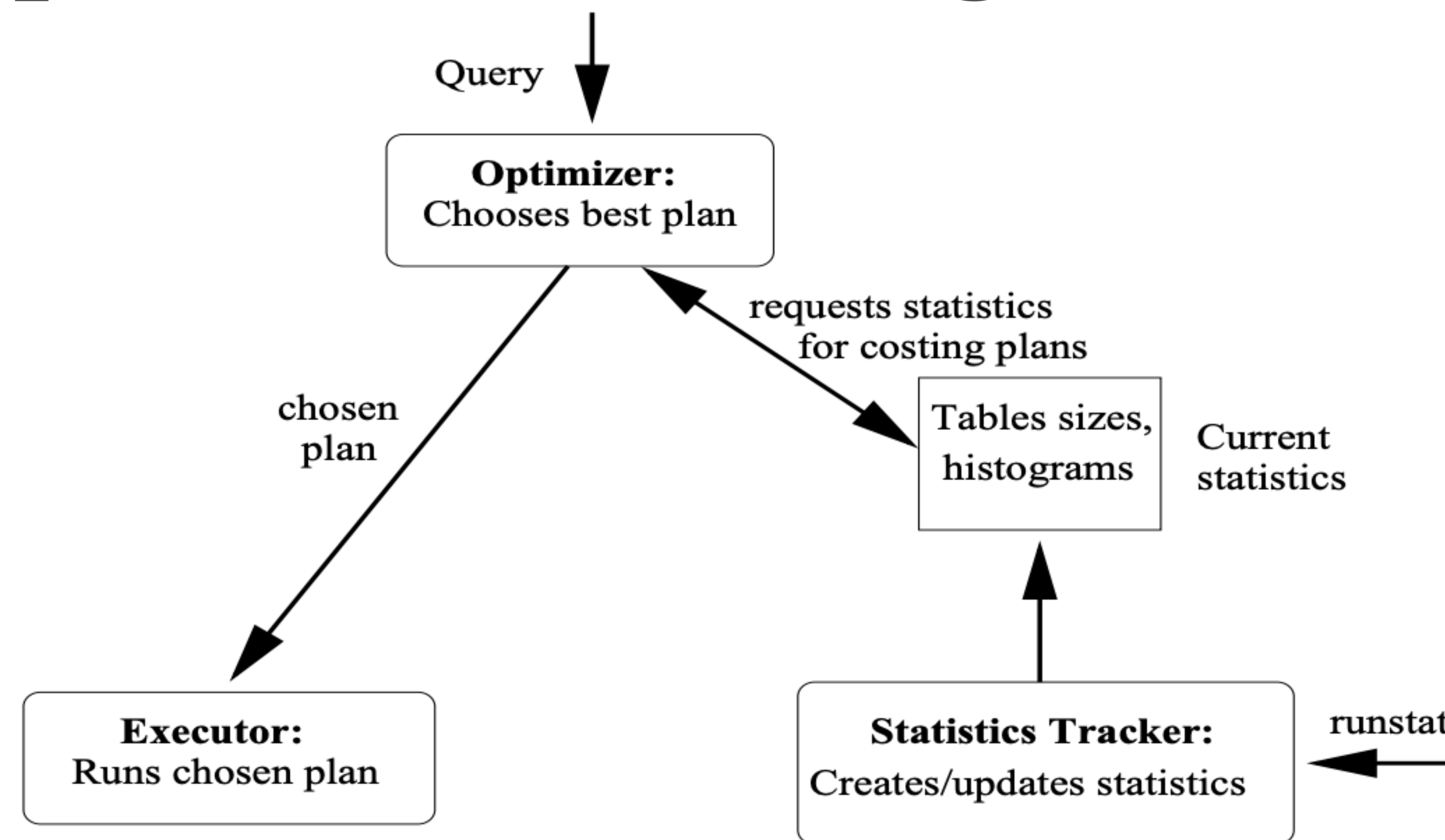


Figure 2: The traditional approach to query processing

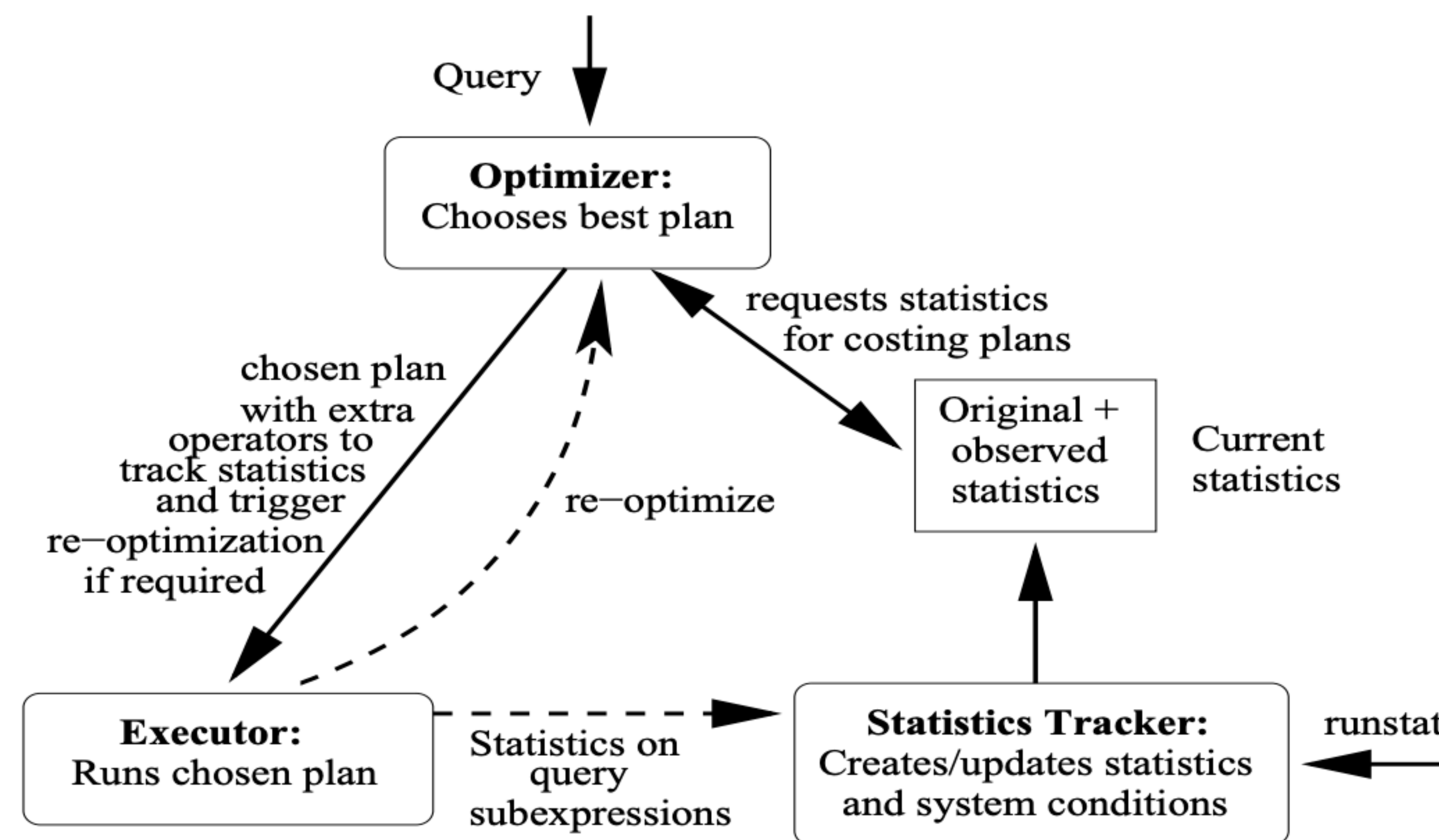


Figure 3: AQP in plan-based systems

如何打造极速数据库 5：Adaptive & AI Tuning

- 优化器多次重新 Plan
- 执行层动态改变执行策略
- 不同的数据分组使用不同的Plan

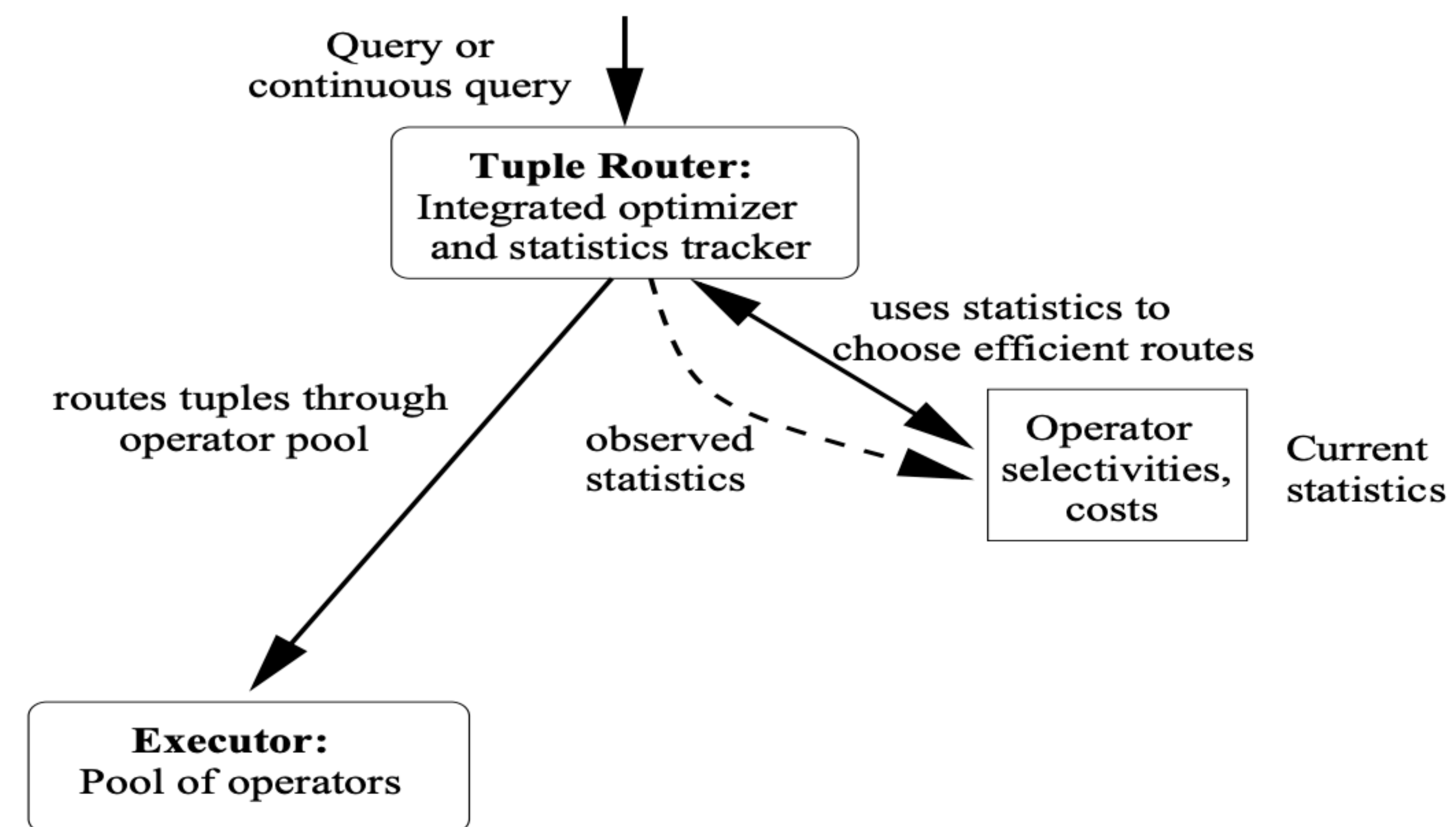


Figure 4: AQP in routing-based systems

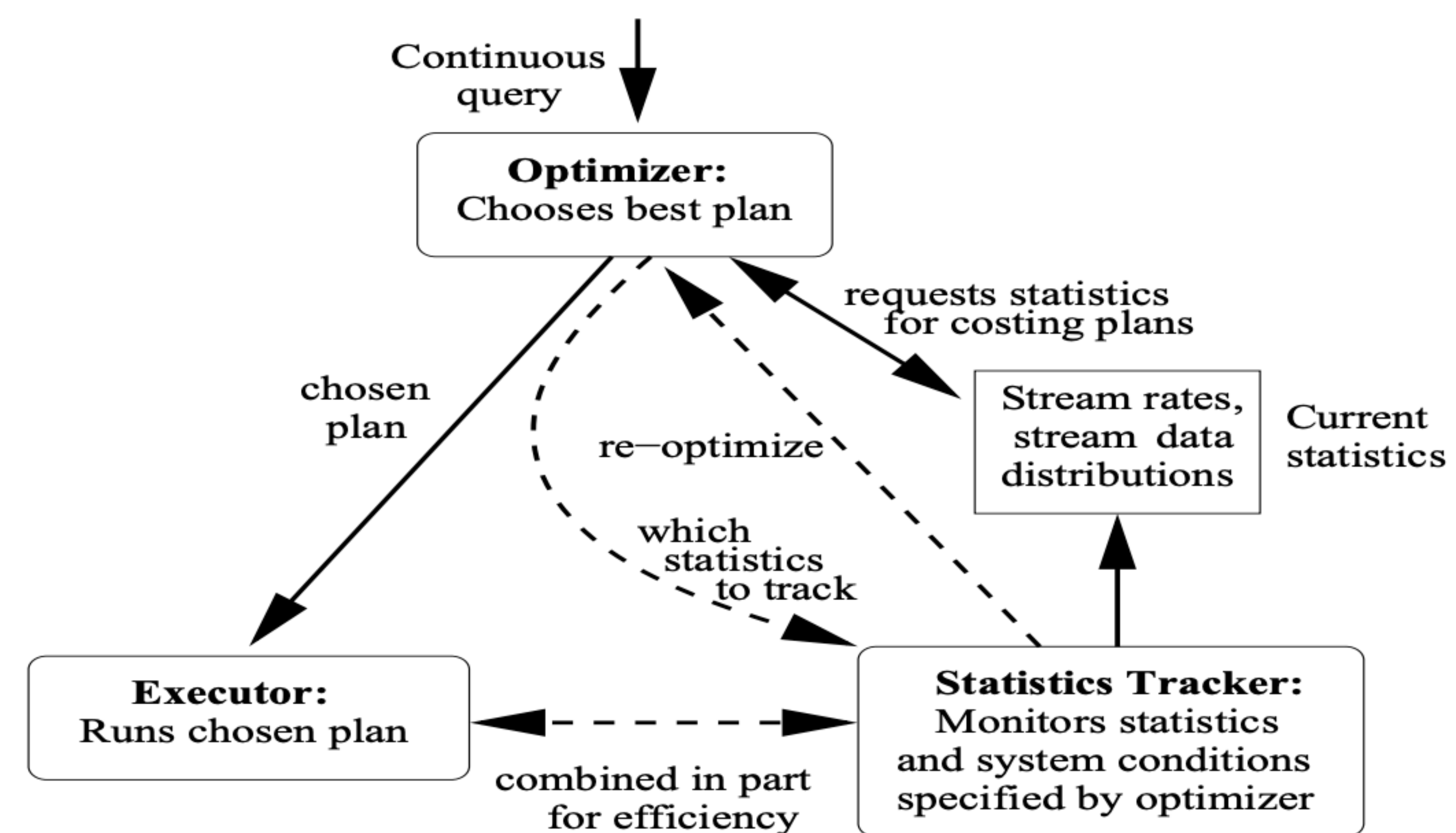


Figure 5: AQP in CQ-based systems

如何打造极速数据库 5：谓词自适应选择

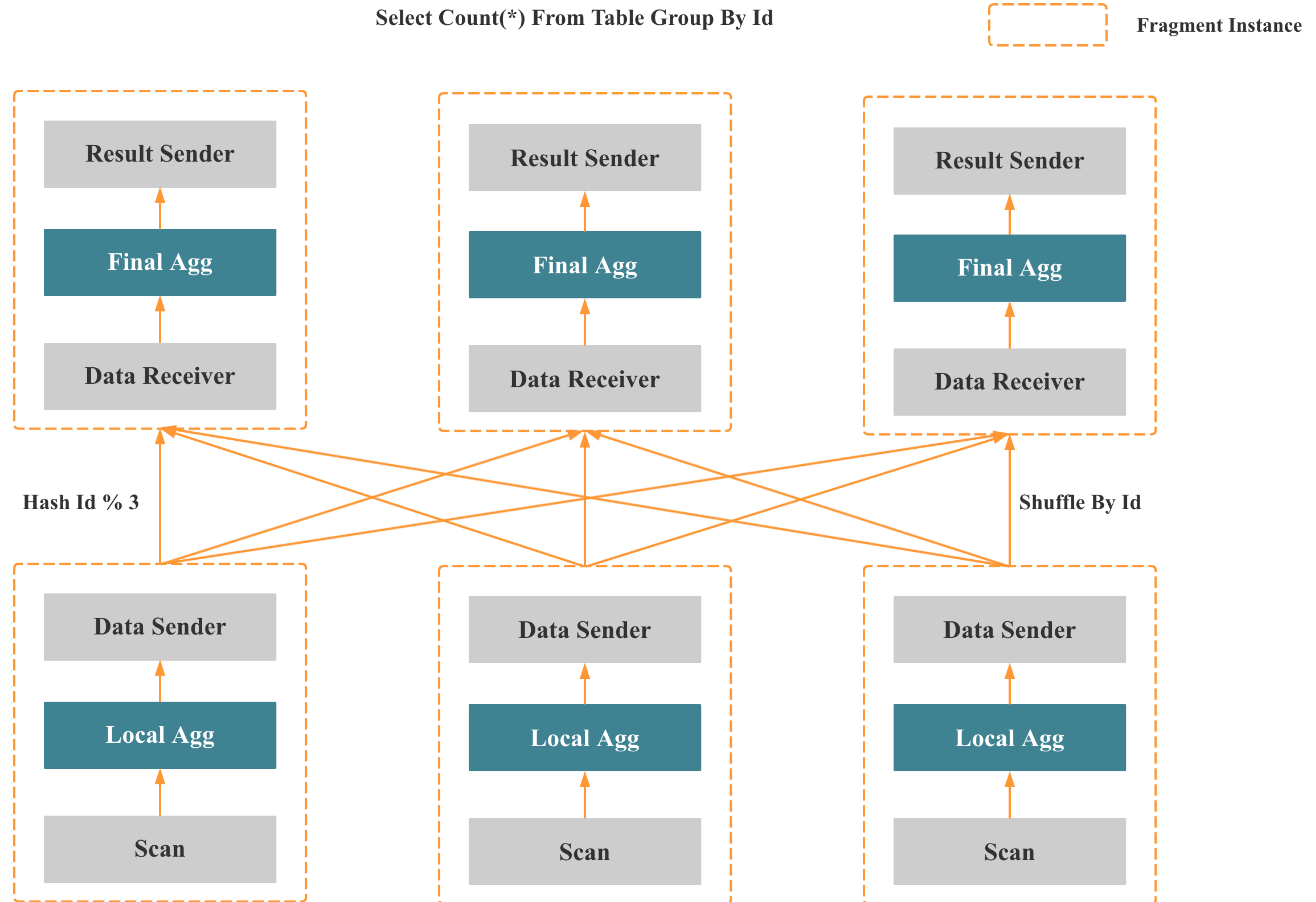
```
double selectivity = true_count * 1.0 / chunk_size;
if (selectivity <= 0.5) {     // useful filter
    if (selectivity < 0.05) { // very useful filter, could early return
        _selectivity.clear();
        _selectivity.emplace(selectivity, rf_desc);
        chunk->filter(new_selection);
        return;
    }

    // Only choose three most selective runtime filters
    if (_selectivity.size() < 3) {
        _selectivity.emplace(selectivity, rf_desc);
    } else {
```

Prefer The Low Selectivity Filter

如何打造极速数据库 5：自适应 Hash 聚合

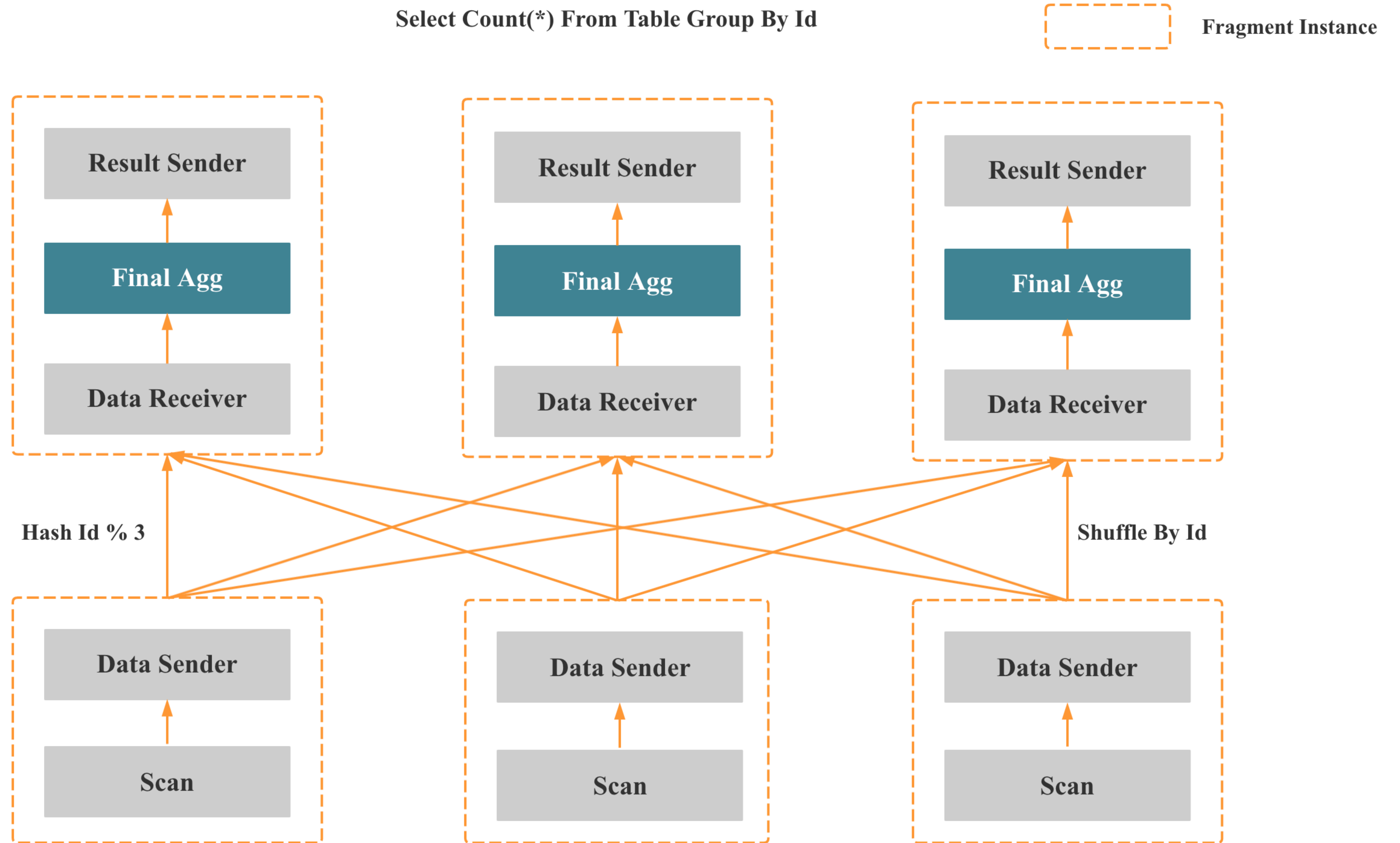
- 分布式两阶段聚合
- Local Agg: 部分聚合
- Final Agg: 全量聚合

Local Agg 需要有聚合效果
可以减少 Shuffle 数据量



如何打造极速数据库 5：自适应 Hash 聚合

- 分布式一阶段聚合
- Final Agg: 全量一次聚合

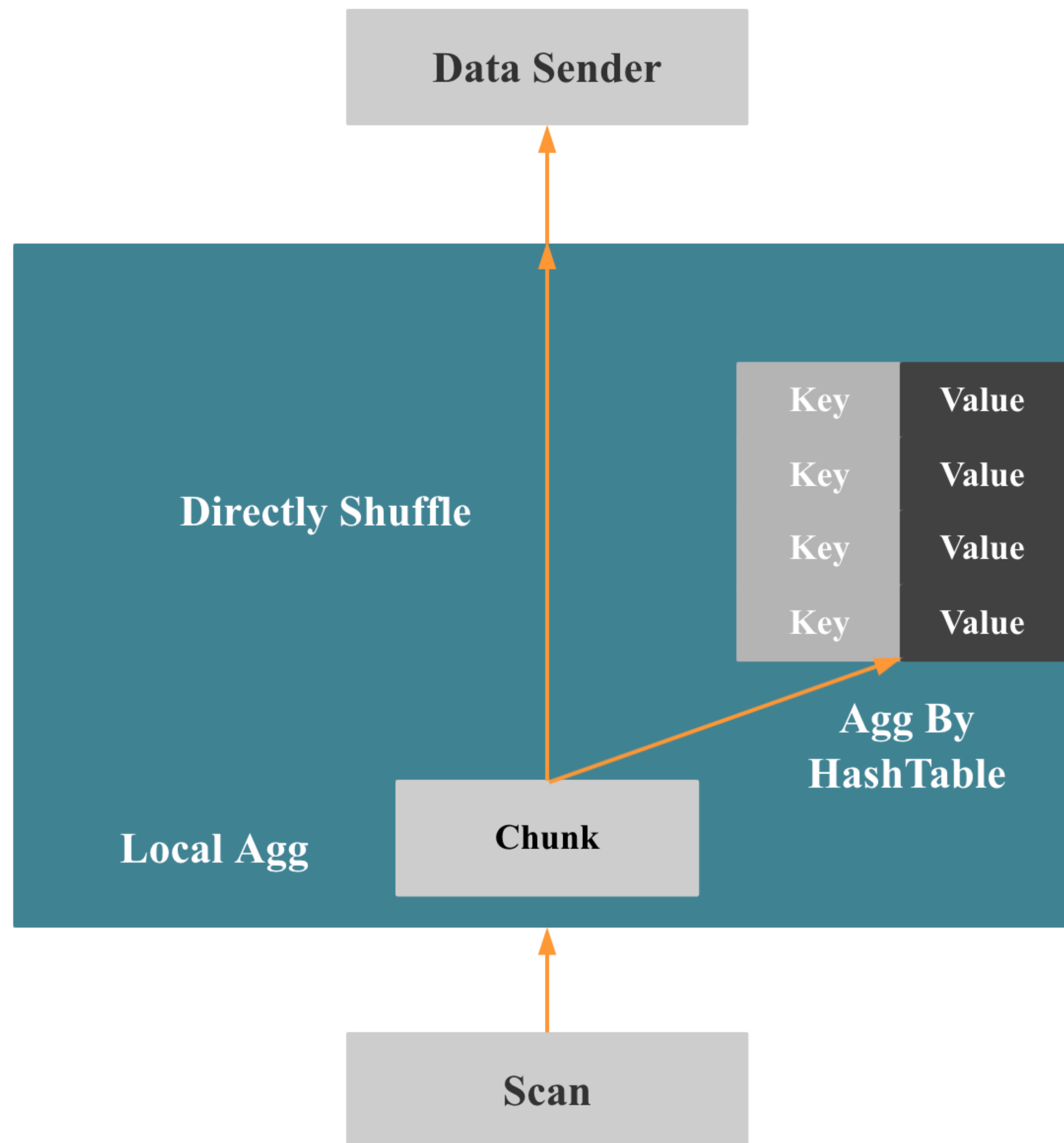


Local Agg 无聚合效果

减少无用 HashTable 的构建

如何打造极速数据库 5：自适应 Hash 聚合

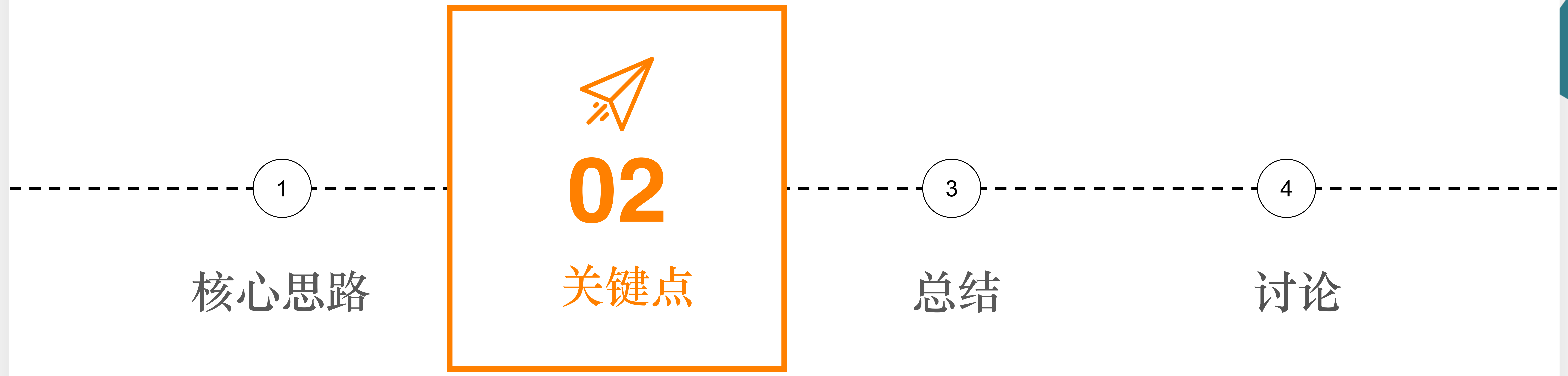
- 运行时自适应
- 有聚合效果走 HashTable
- 无聚合效果直接 Shuffle



如何打造极速数据库：总结

- 预处理 (Materialized View)
- 扩展性：多核，多机，弹性伸缩
- 数据流：优化器，调度器，执行器，存储引擎
- Adaptive & AI Tuning
- 近似计算
- 工程细节
- 与易用性，可观测性等的权衡

如何打造一款极速数据库

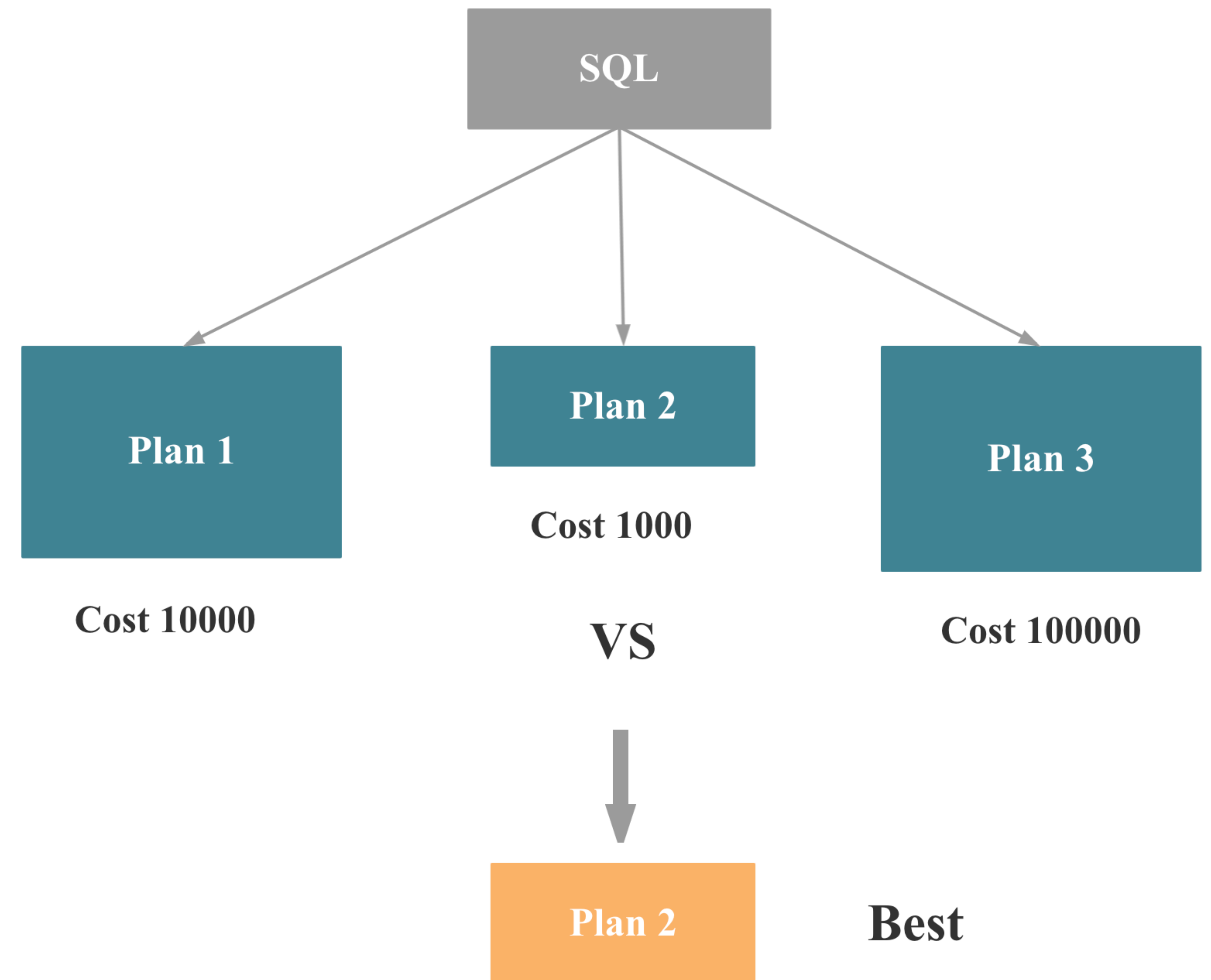


StarRocks 极速关键点

1. CBO 优化器
2. 查询调度器
3. MPP 分布式执行框架
4. Pipeline 多核并行框架
5. 向量化执行器
6. 列式存储

CBO 查询优化器

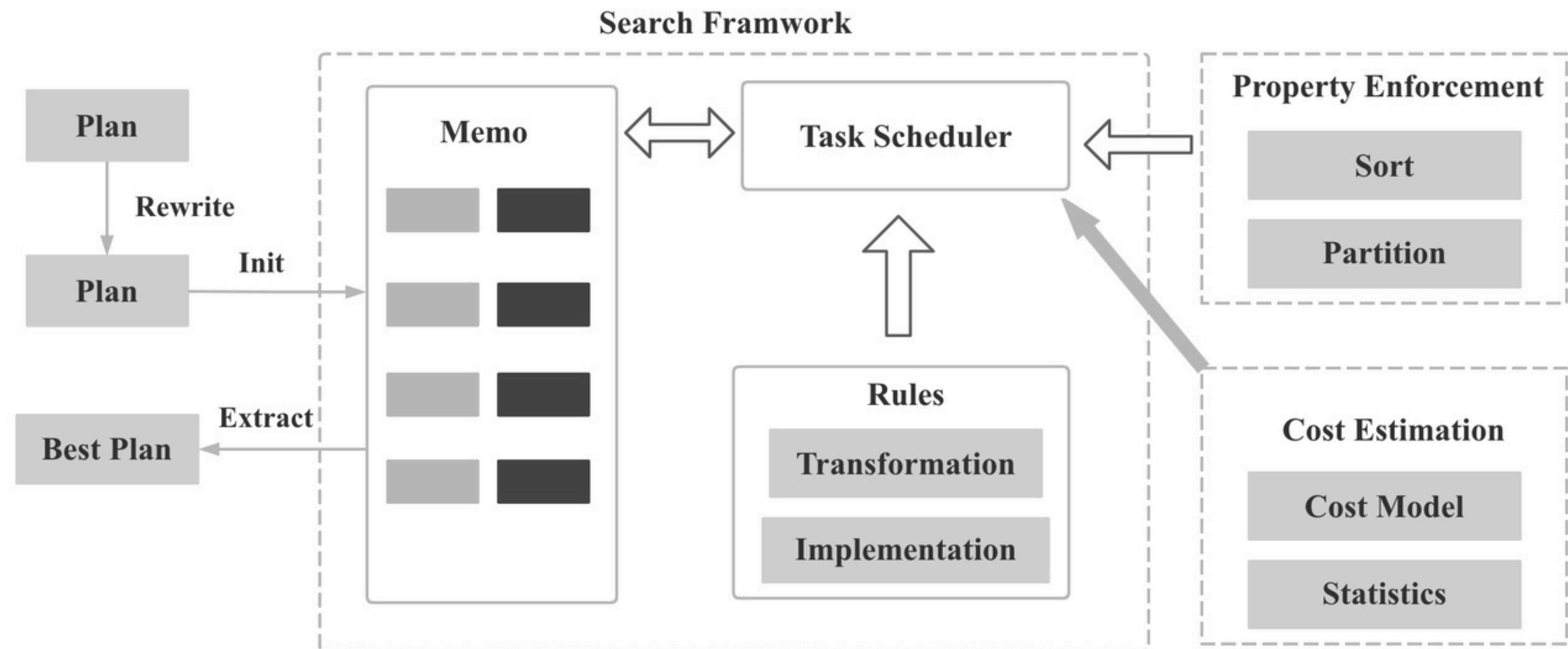
- 输入： Logical Plan
- 输出： Cost 最小的分布式物理执行计划



StarRocks CBO 查询优化器

- 完全自研
- 基于 Cascades 和 ORCA 论文
- 结合 StarRocks 执行器和调度器

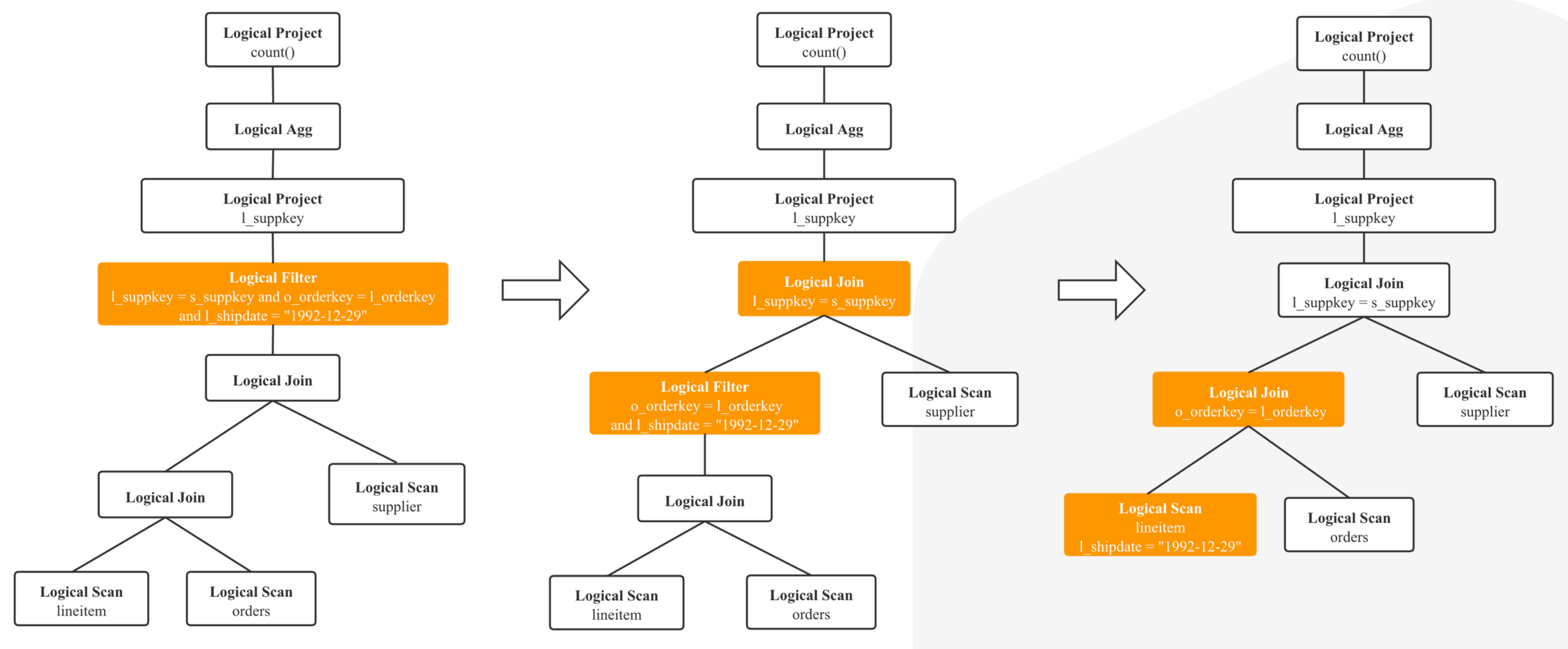
  深度定制，优化和创新



StarRocks CBO 查询优化器：Logical Plan Rewrite

- 各种表达式的重写和化简
- 列裁剪
- 谓词下推
- Limit Merge, Limit 下推
- 聚合 Merge
- 等价谓词推导（常量传播）
- Outer Join 转 Inner Join
- Intersect Reorder

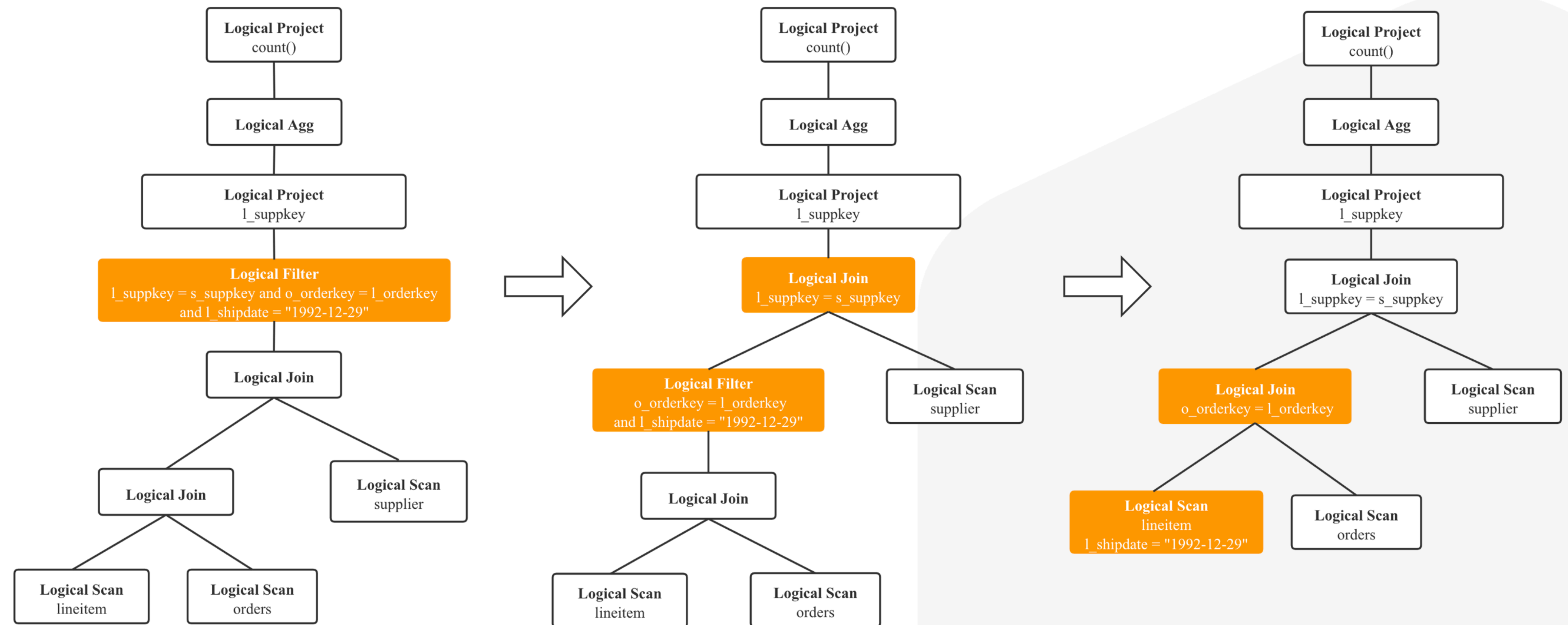
SELECT count(*)
FROM lineitem, supplier, orders
WHERE l_suppkey = s_suppkey AND o_orderkey = l_orderkey and l_shipdate = “1992-12-29”;



StarRocks CBO 查询优化器：Logical Plan Rewrite

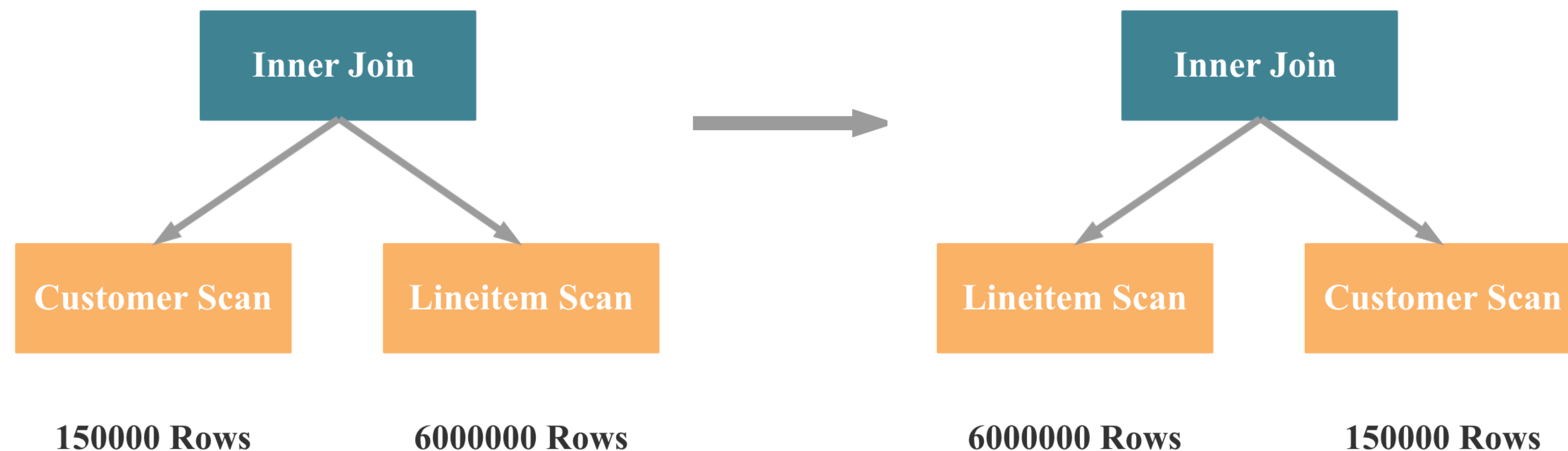
- 常量折叠
- 公共表达式复用
- 子查询重写
- Lateral Join 化简
- 分区分桶裁剪
- Empty Node 优化
- Empty Union, Intersect, Except 裁剪
- Count Distinct 相关聚合函数重写

```
SELECT count(*)
FROM lineitem, supplier, orders
WHERE l_suppkey = s_suppkey AND o_orderkey = l_orderkey and l_shipdate = “1992-12-29”;
```



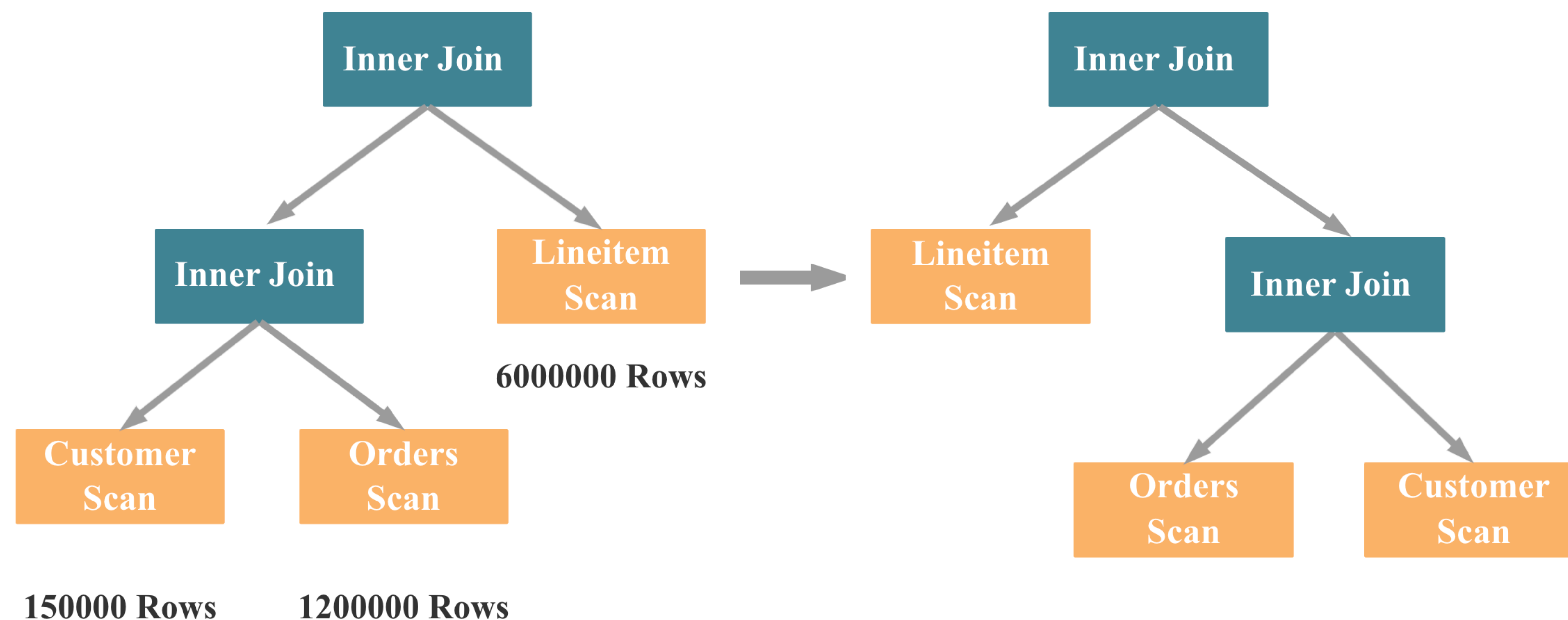
StarRocks CBO 查询优化器: CBO Transform

- 多阶段聚合优化
- **Join 左右表调整**
- Join 多表 Reorder
- Join 分布式执行选择
- 物化视图选择与重写



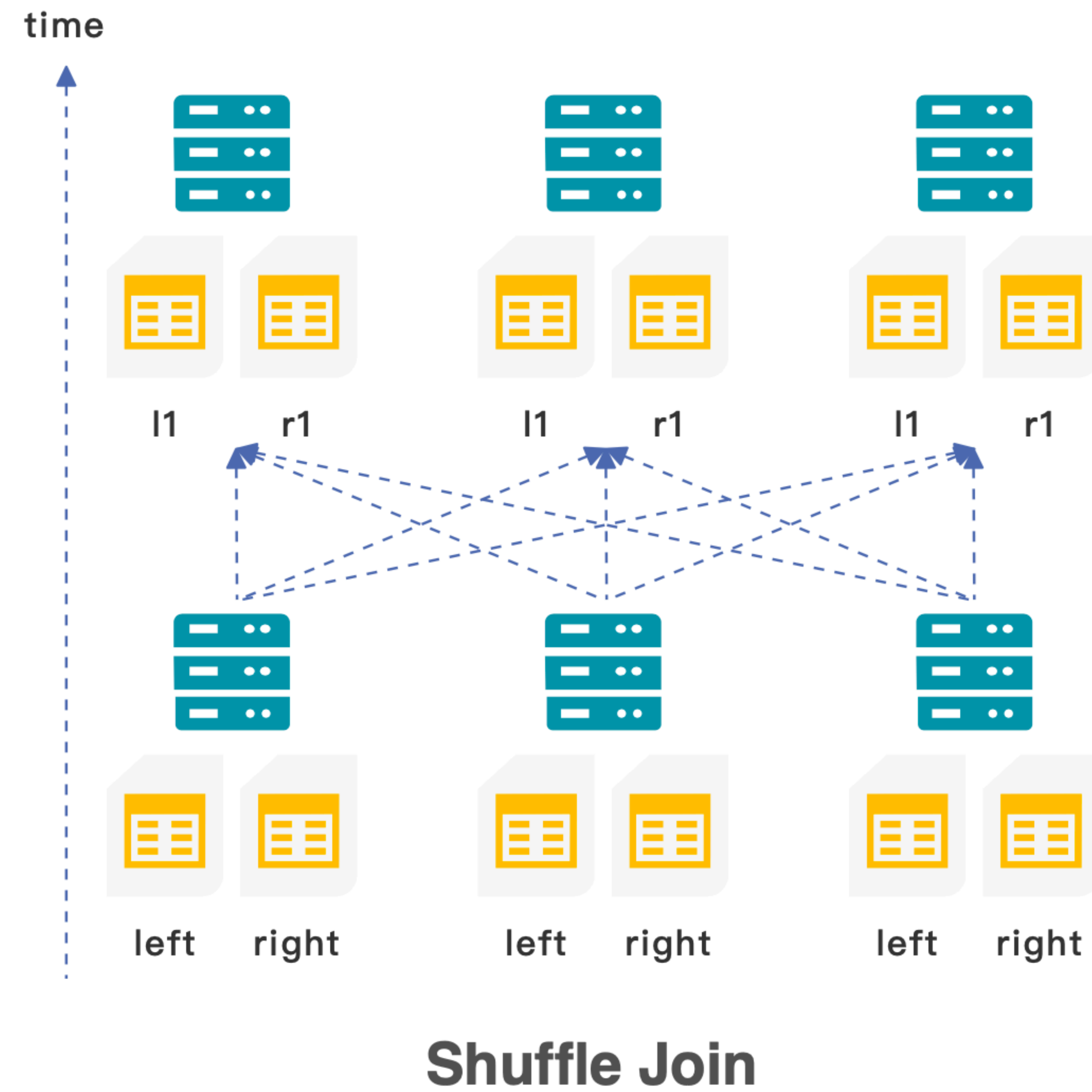
StarRocks CBO 查询优化器: CBO Transform

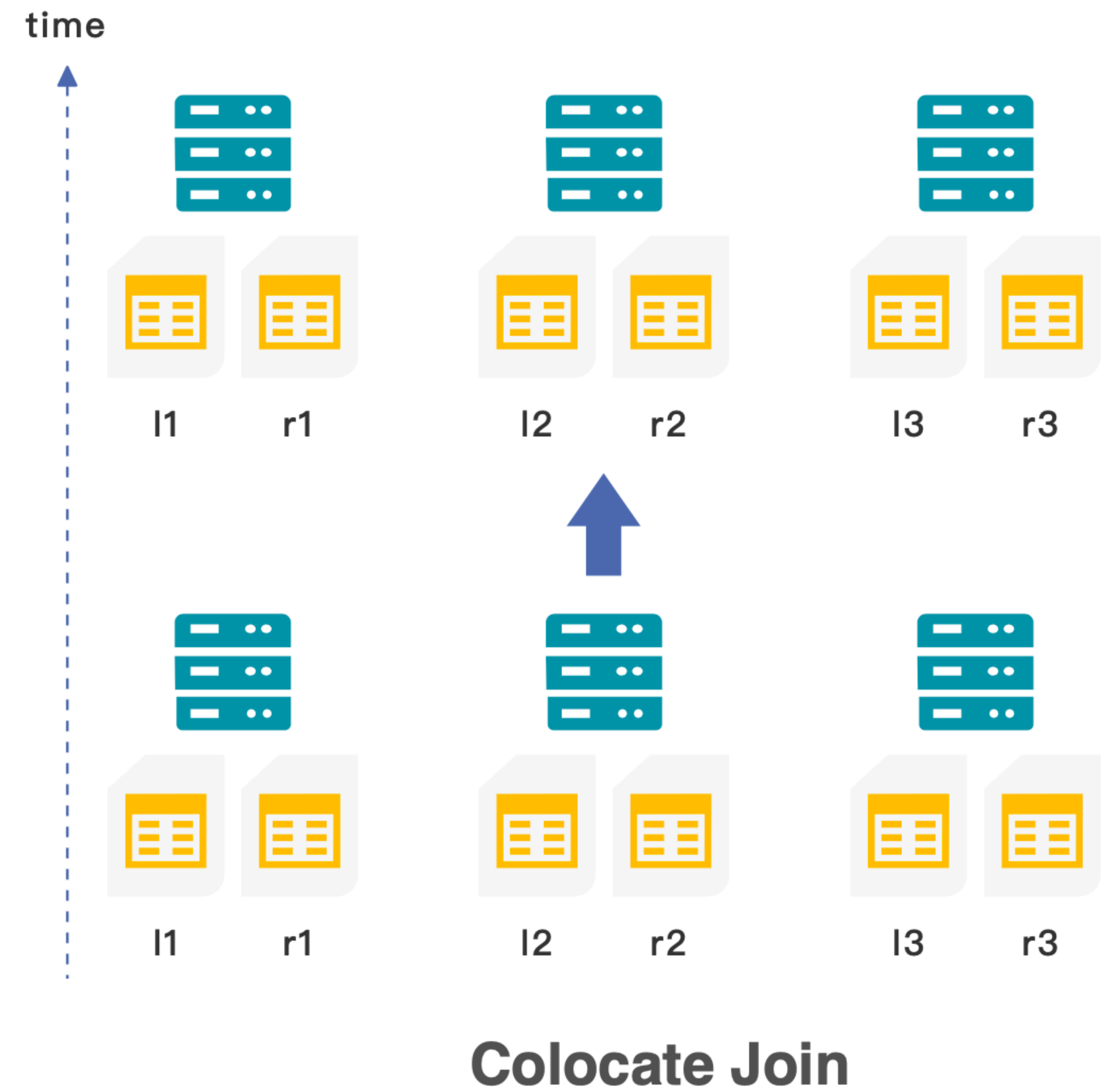
- 多阶段聚合优化
- Join 左右表调整
- **Join 多表 Reorder**
- Join 分布式执行选择
- 物化视图选择与重写



StarRocks CBO 查询优化器: CBO Transform

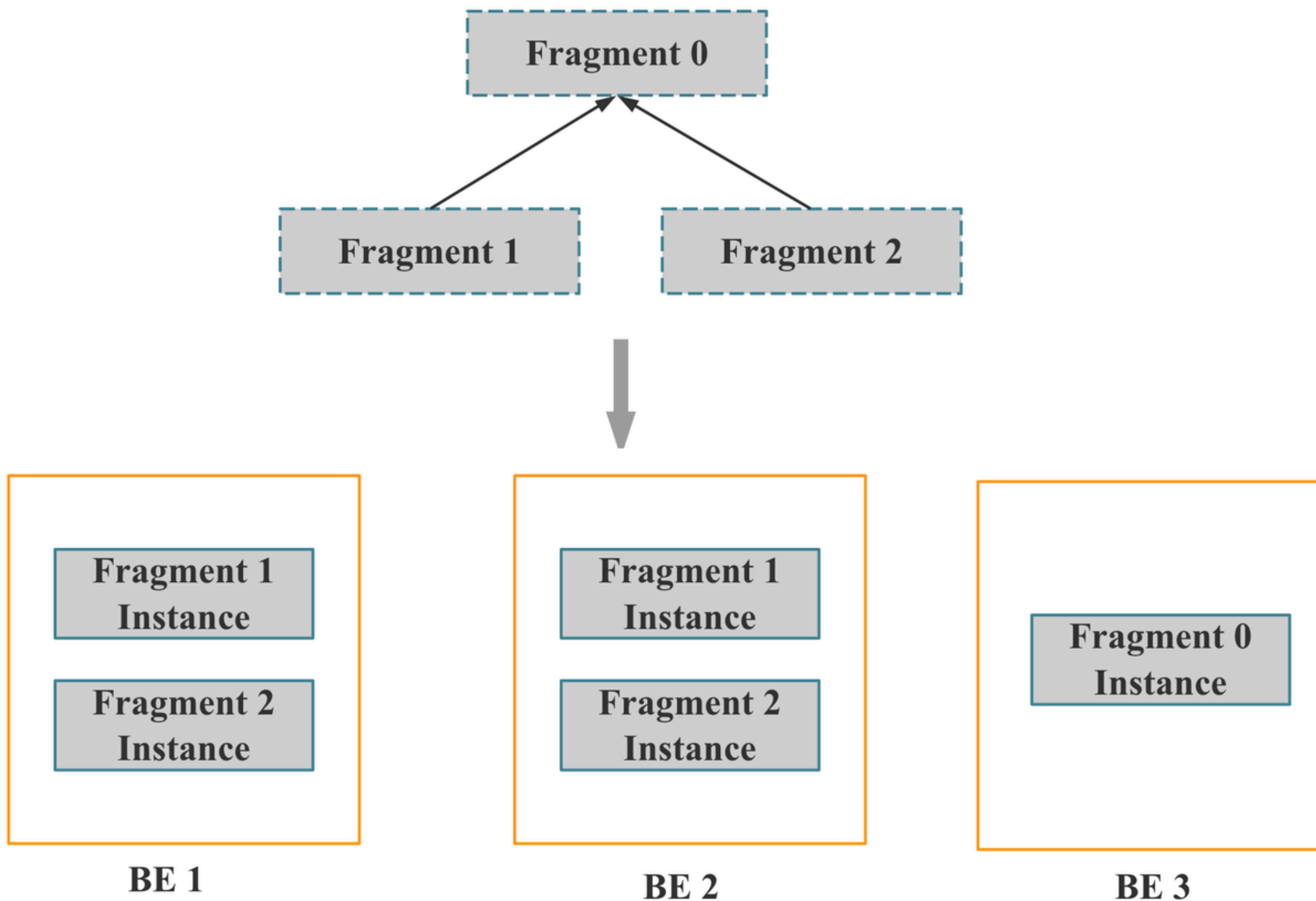
- 多阶段聚合优化
- Join 左右表调整
- Join 多表 Reorder
- **Join 分布式执行选择**
- 物化视图选择与重写





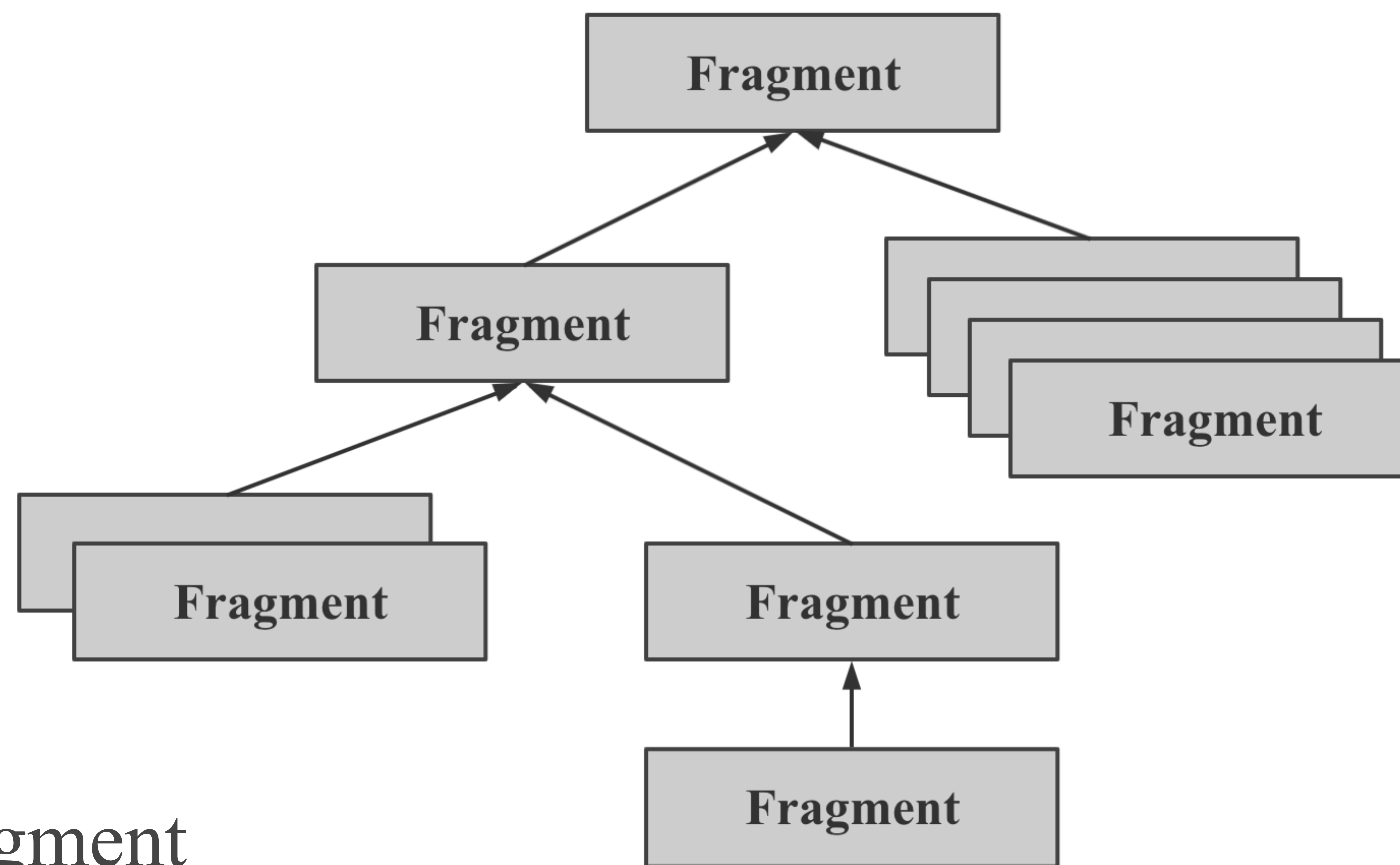
查询调度器

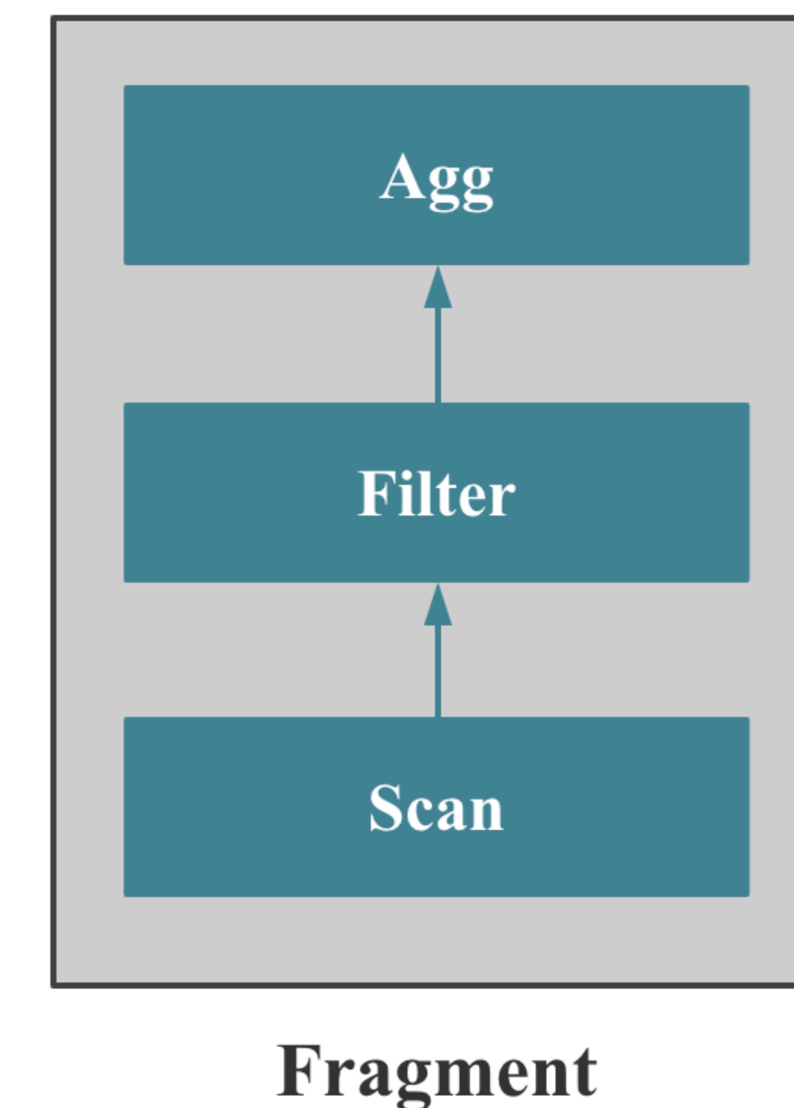
- 执行节点选择策略
- Fragment 分配策略
- Fragment 调度策略



MPP 分布式执行

- Fragment 是 BE 上的执行单元
- Fragment 包括多个 Operator
- Fragment 可以有多个执行实例
- 1个 BE 可以执行多个独立的 Fragment

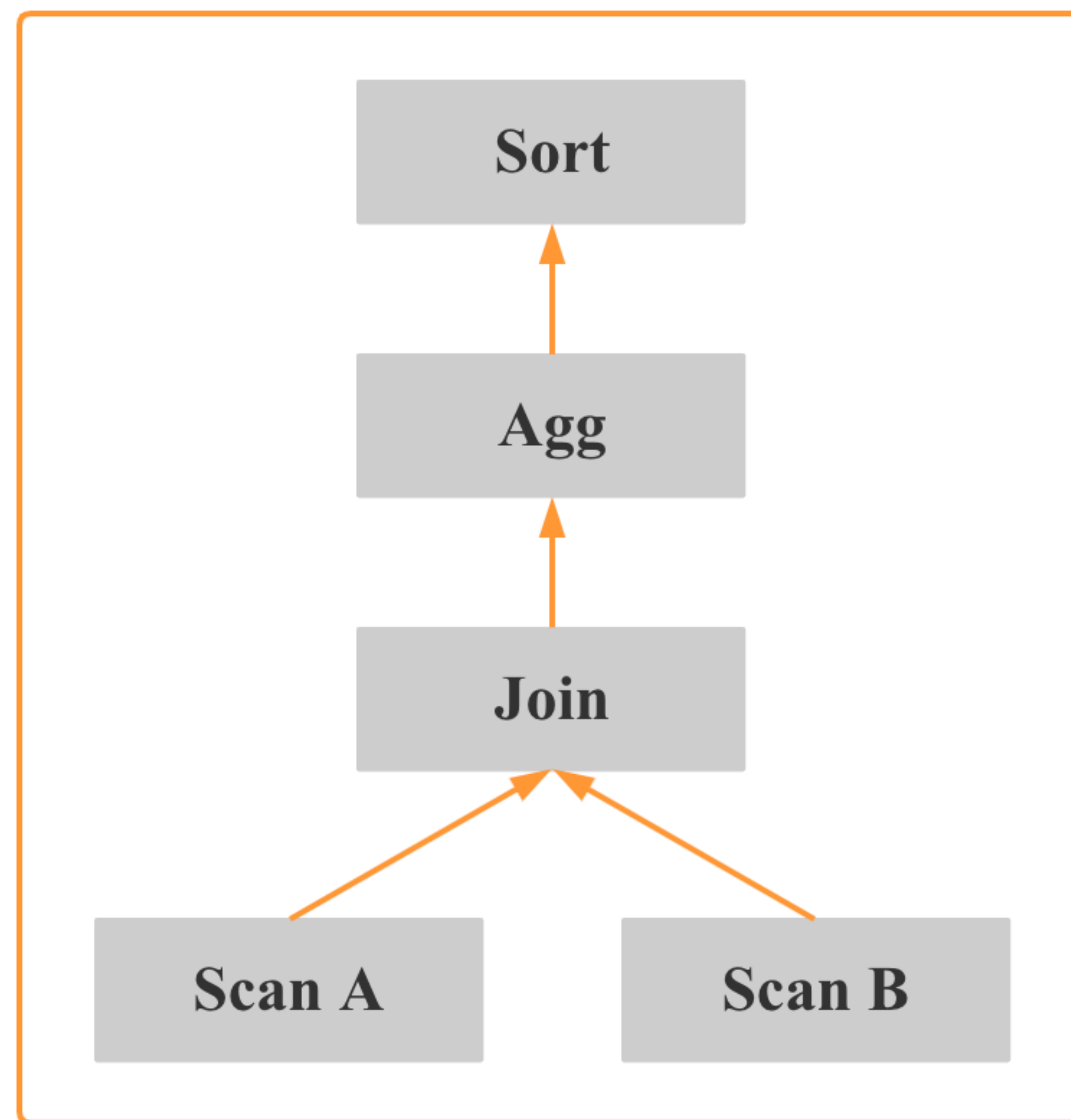




Fragment

MPP 分布式执行

Select K1, Sum(V1)
From A, B
Where A.K2 = B.K2
Group By K1
Order By Sum(V1)



Sort
Agg
Exchange

Sort
Agg
Exchange

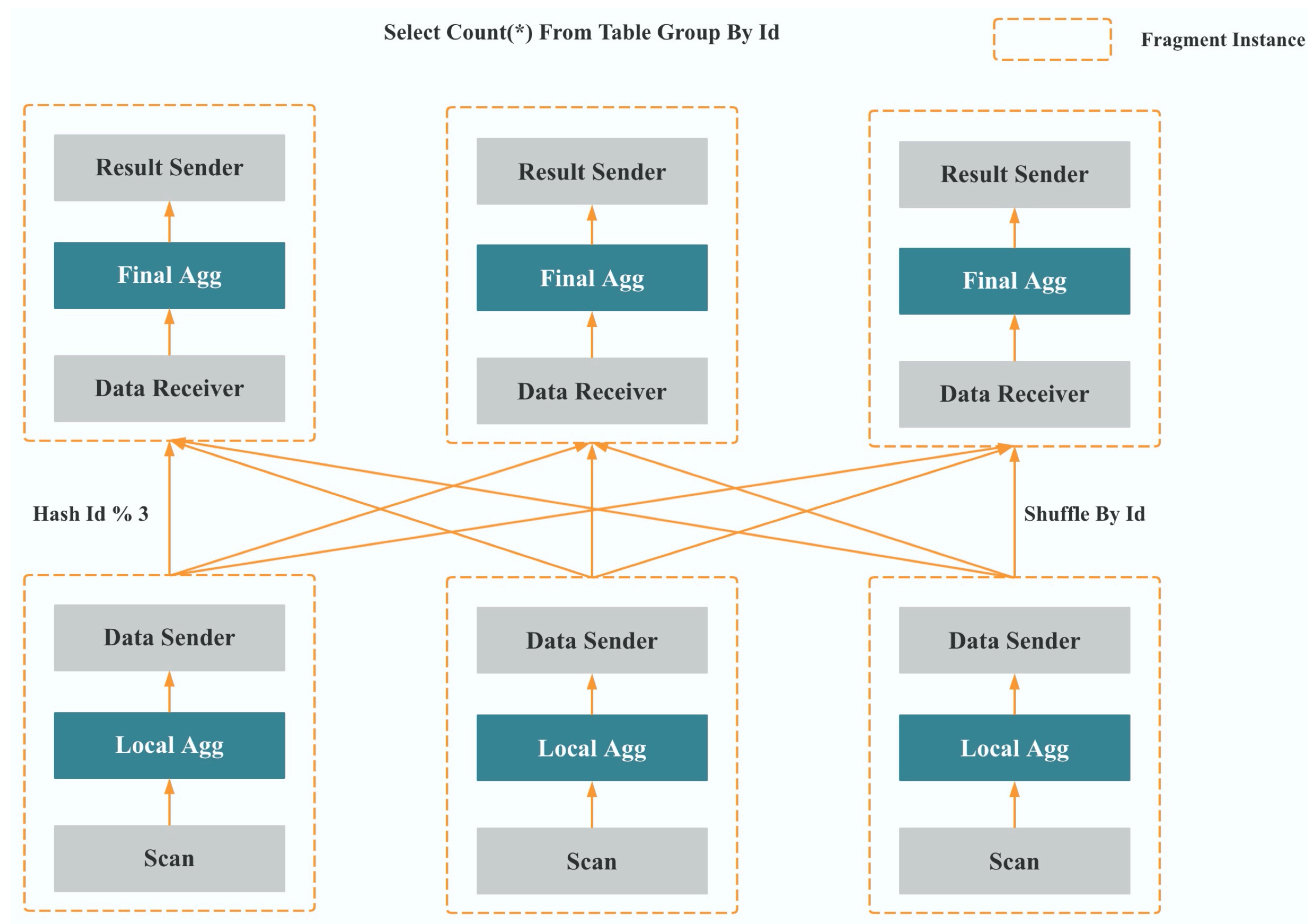
Sort
Agg
Exchange

DataSink
Agg
Join
Scan A
Exchange

DataSink
Scan B

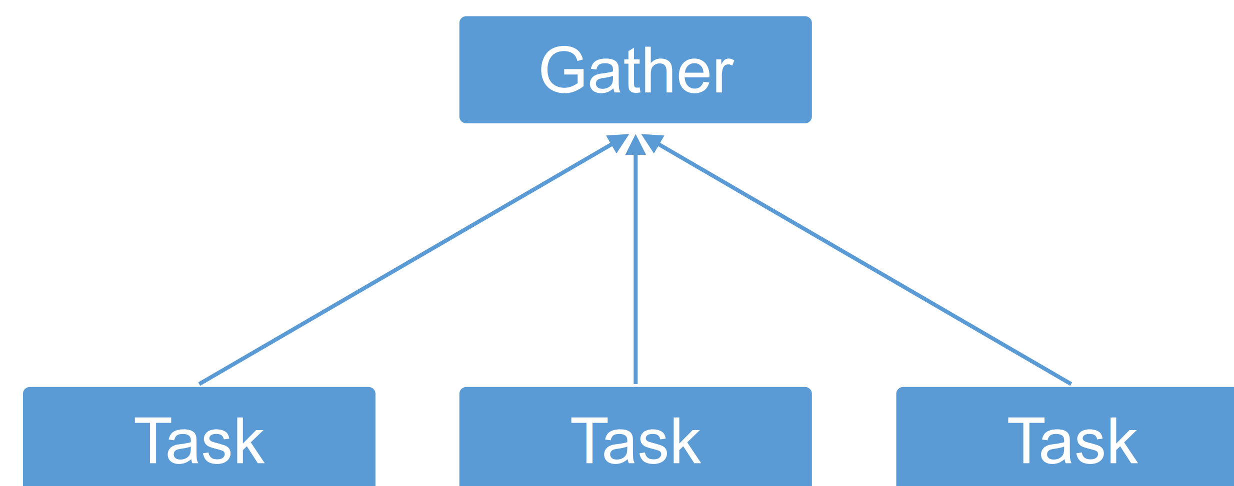
MPP 分布式执行

高基数 Shuffle 能力



MPP 分布式执行

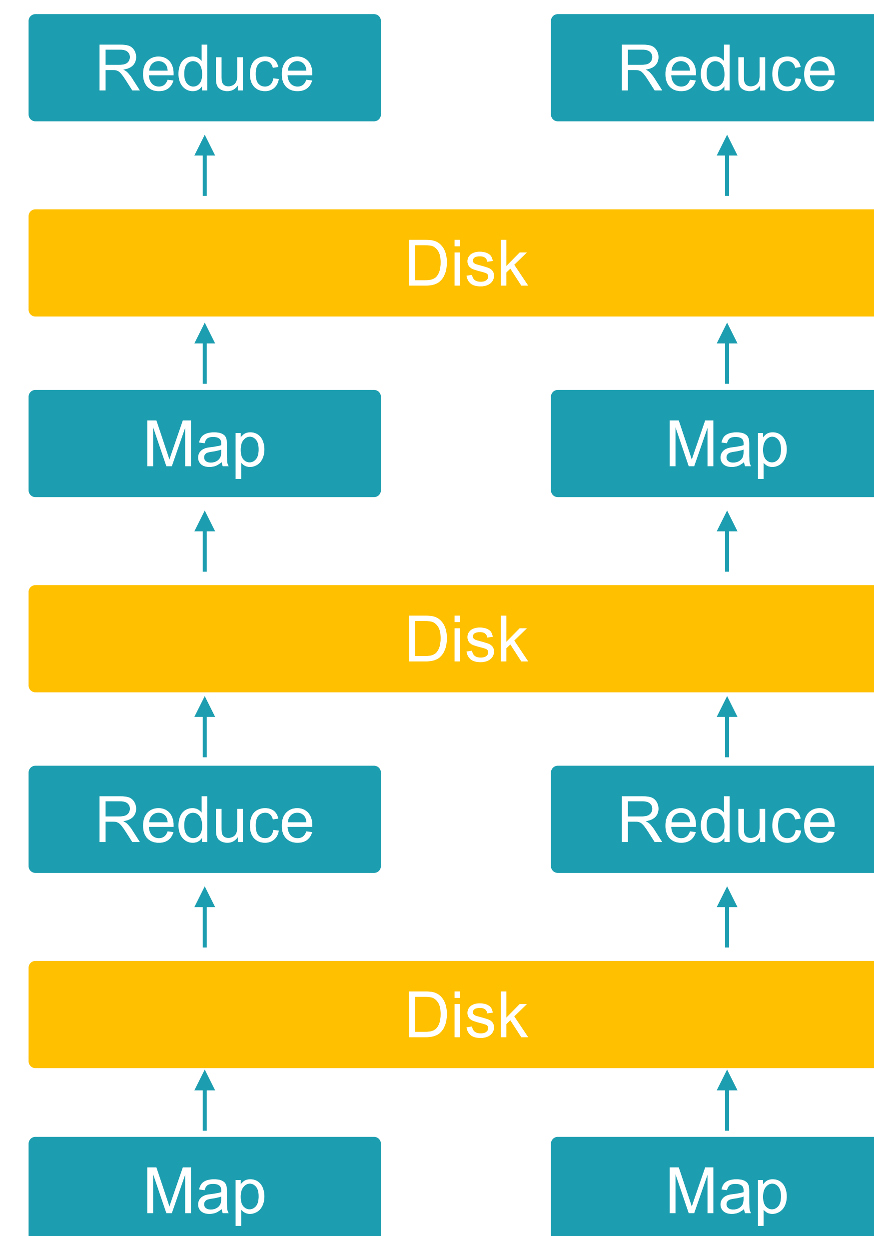
Scatter/Gather



单节点汇聚

无法完成大表Join，高基数聚合

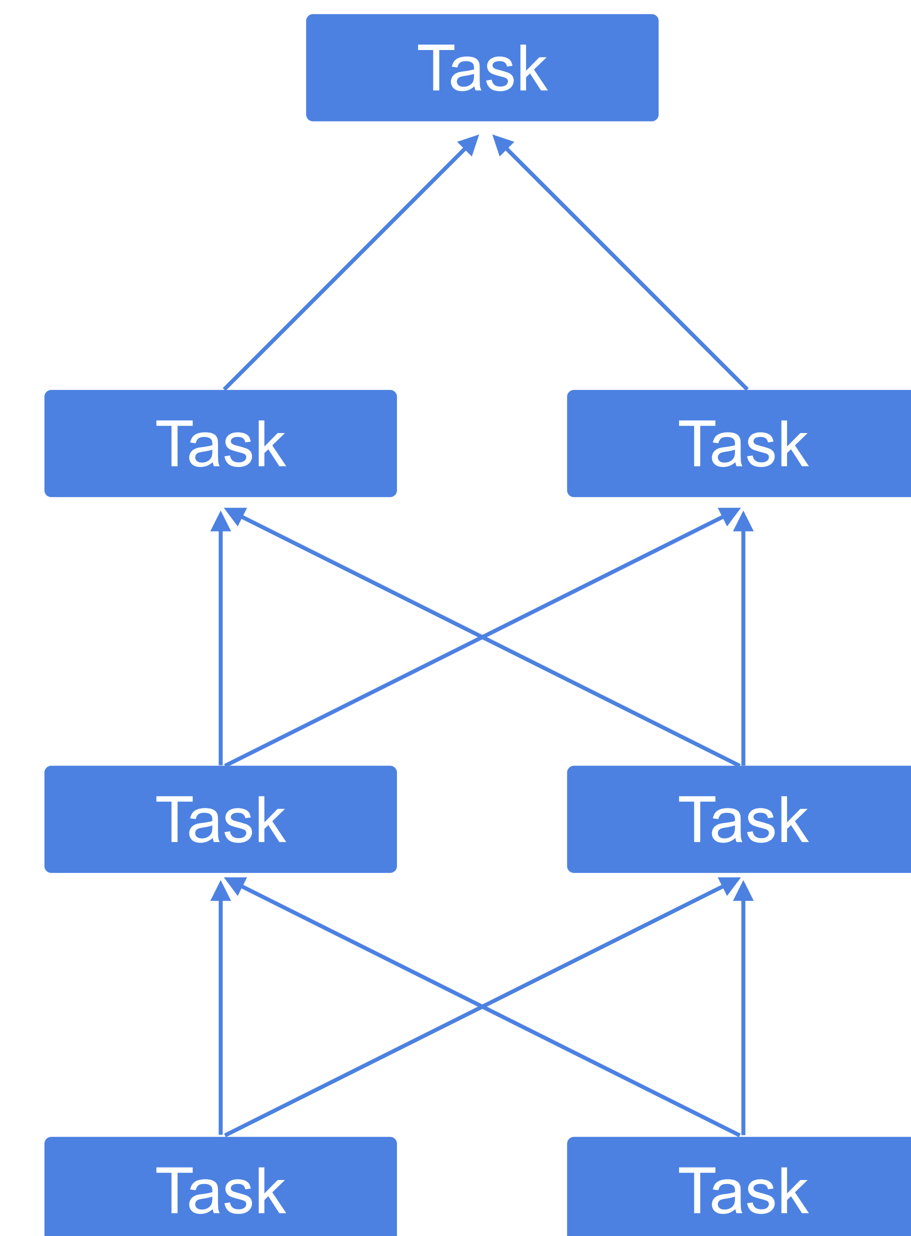
Stage By Stage



任务之间需要等待

中间数据落盘 磁盘IO

MPP



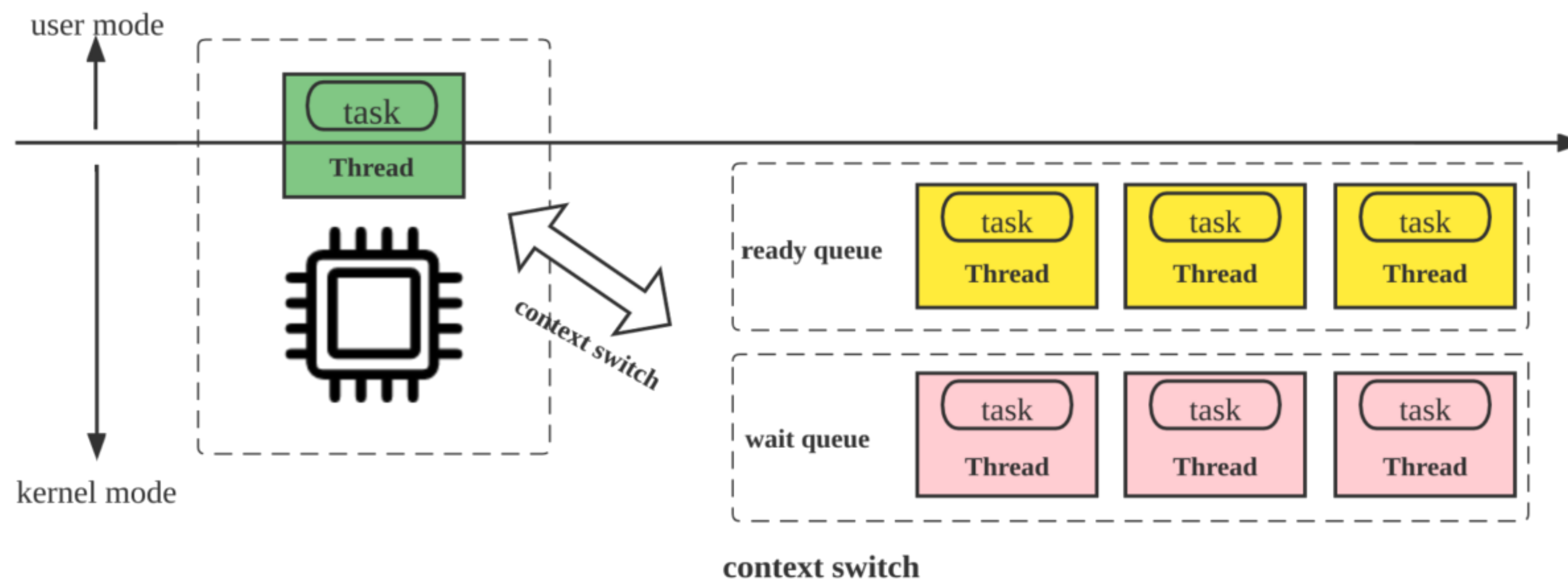
流水线执行无需等待

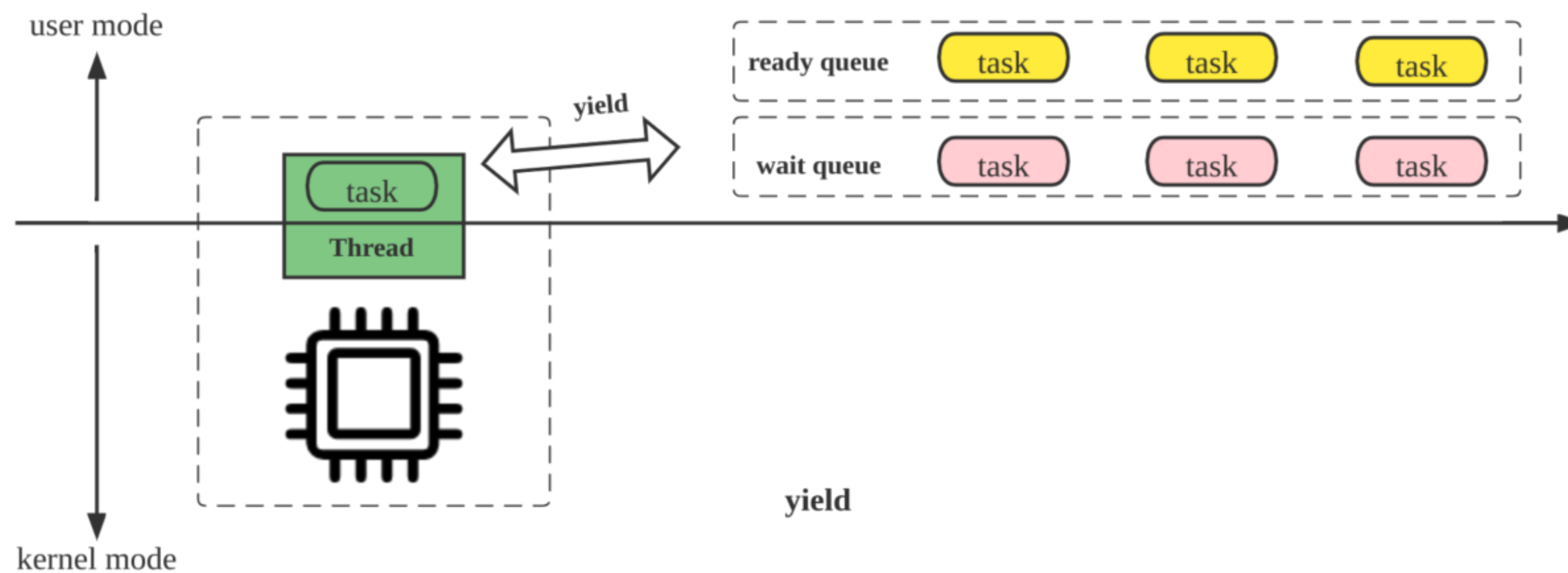
数据内存传输 无磁盘IO

Pipeline 多核并行

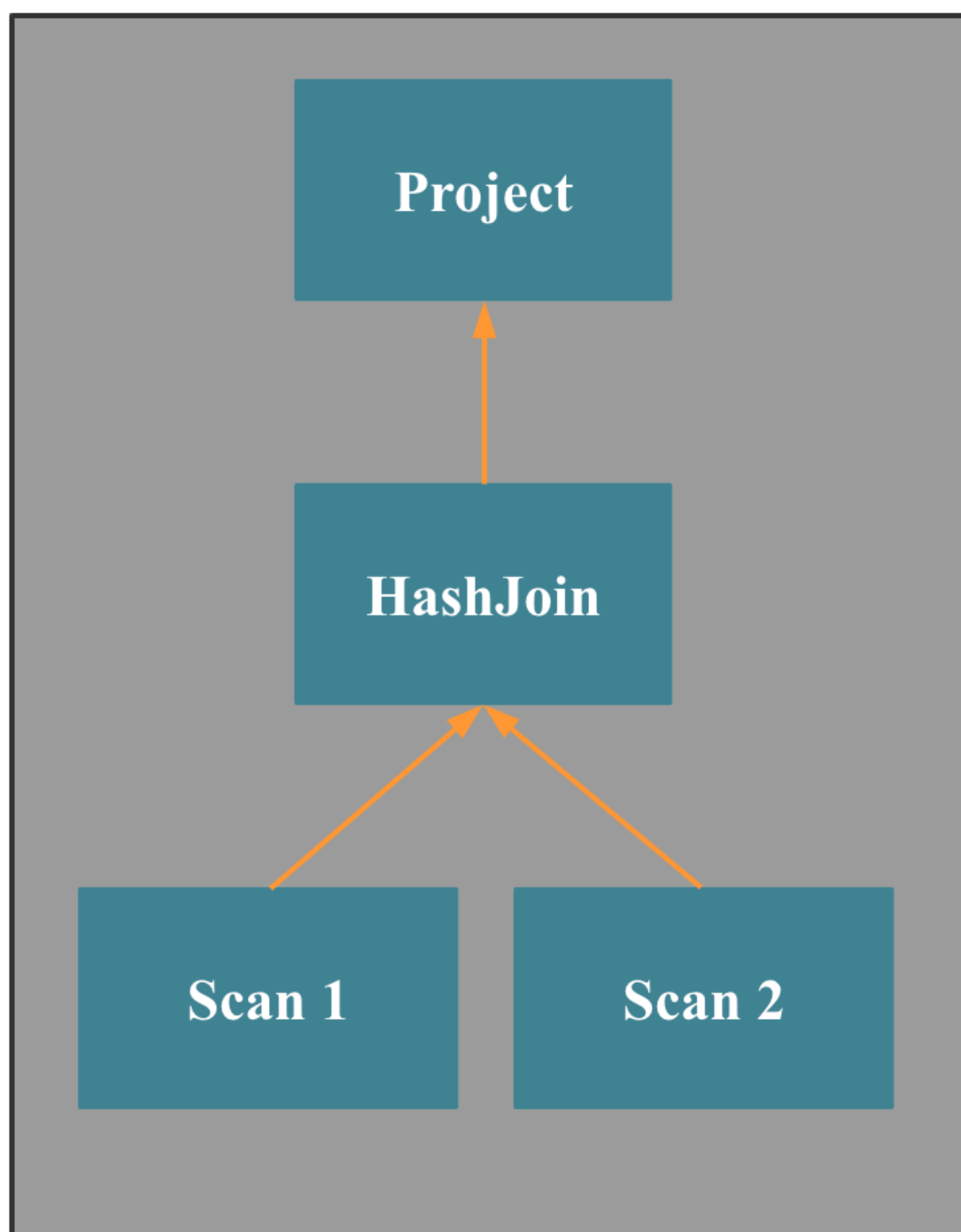
从内核态调度

到用户态调度



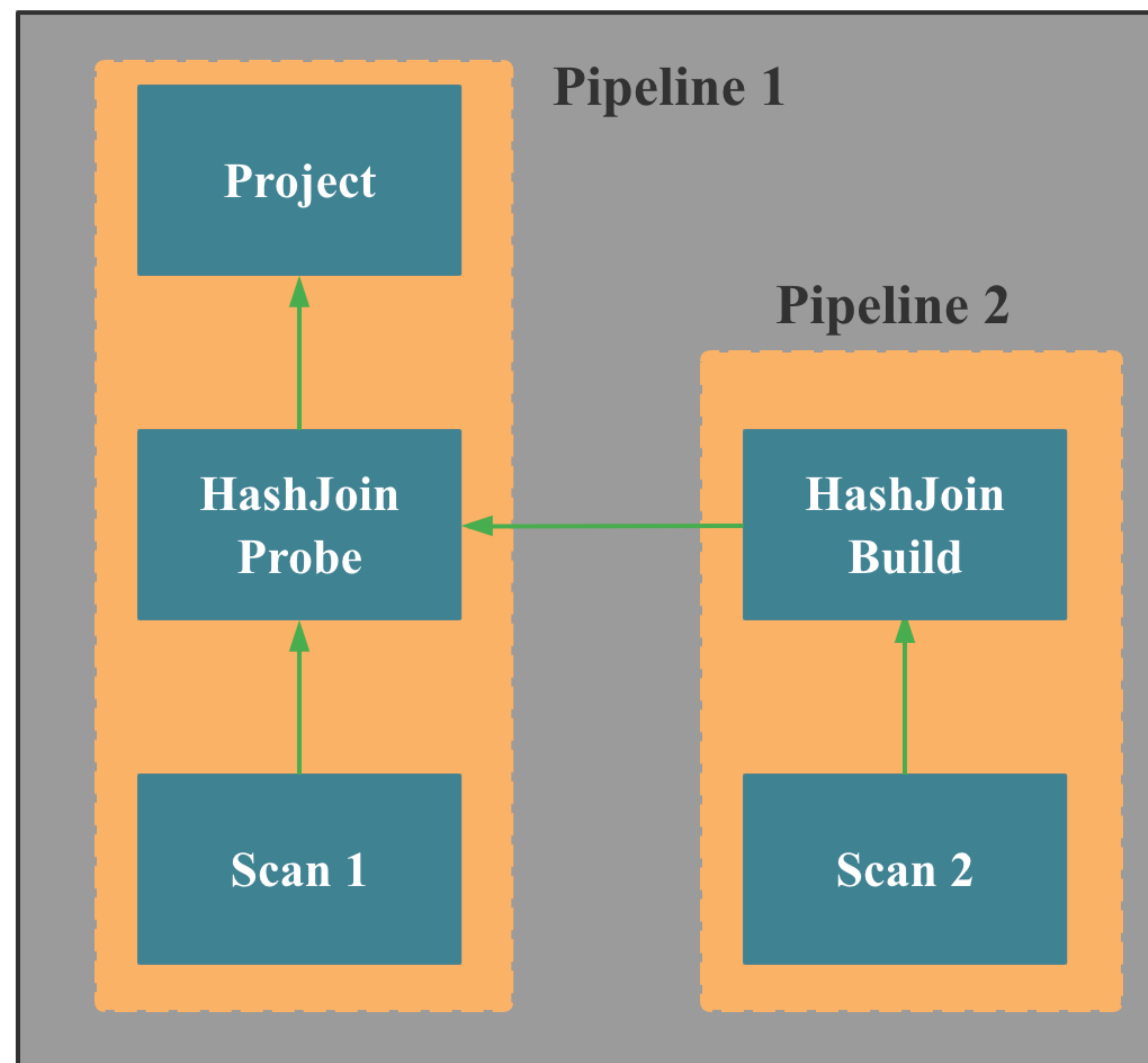


Pipeline 多核并行



Fragment

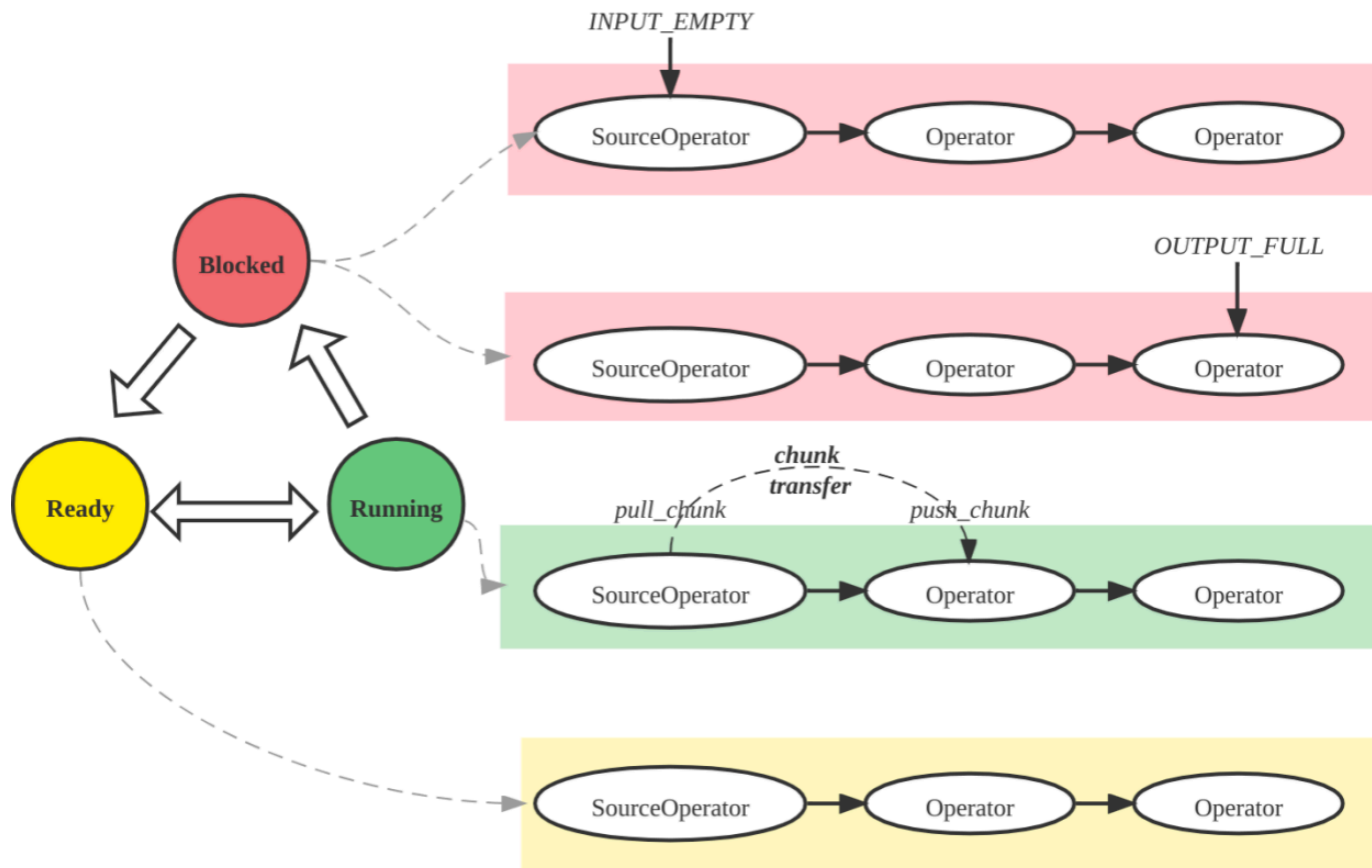
Pipeline



Fragment

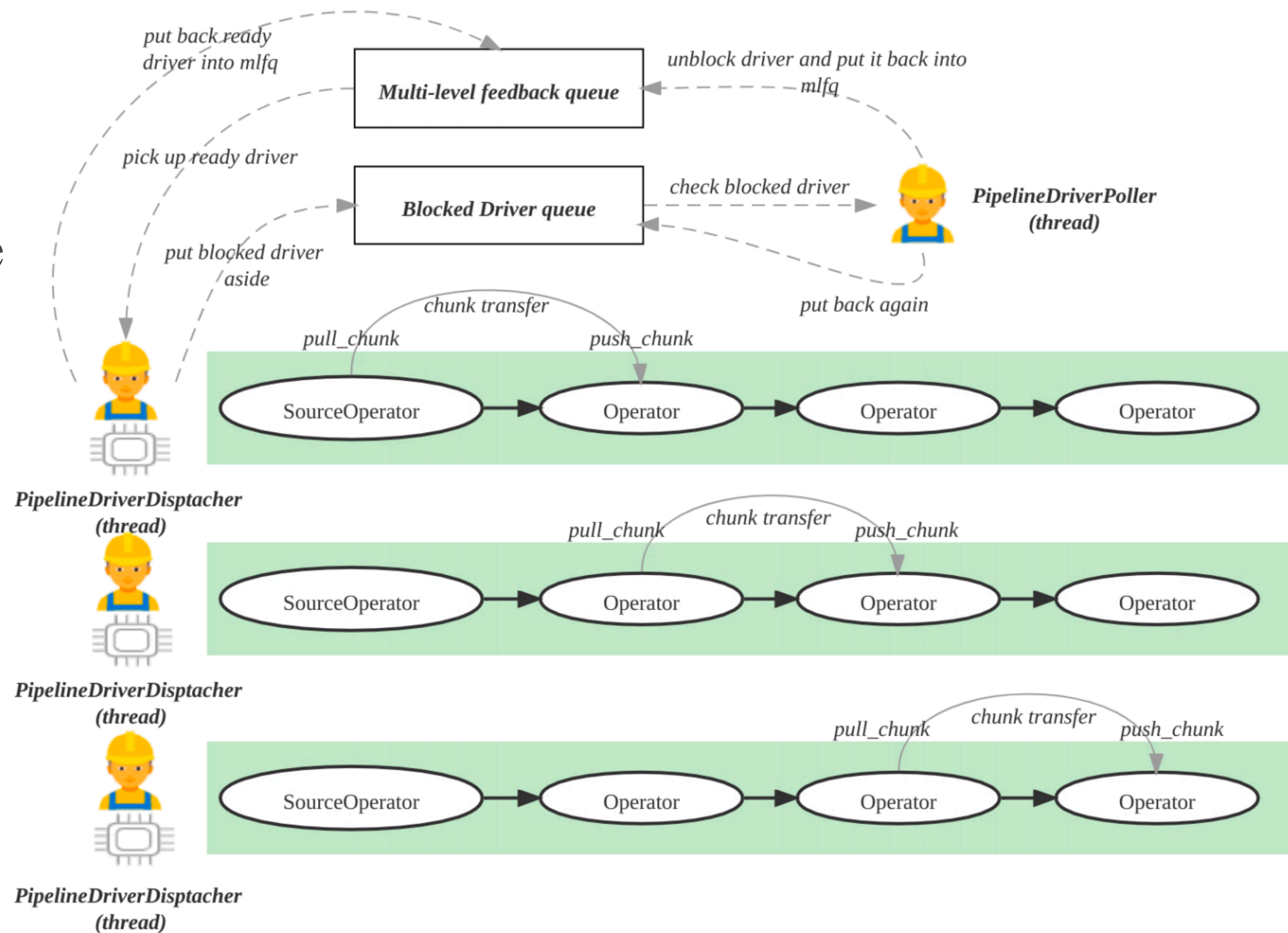
Pipeline: No Materialized

Pipeline 多核并行



Pipeline 多核并行

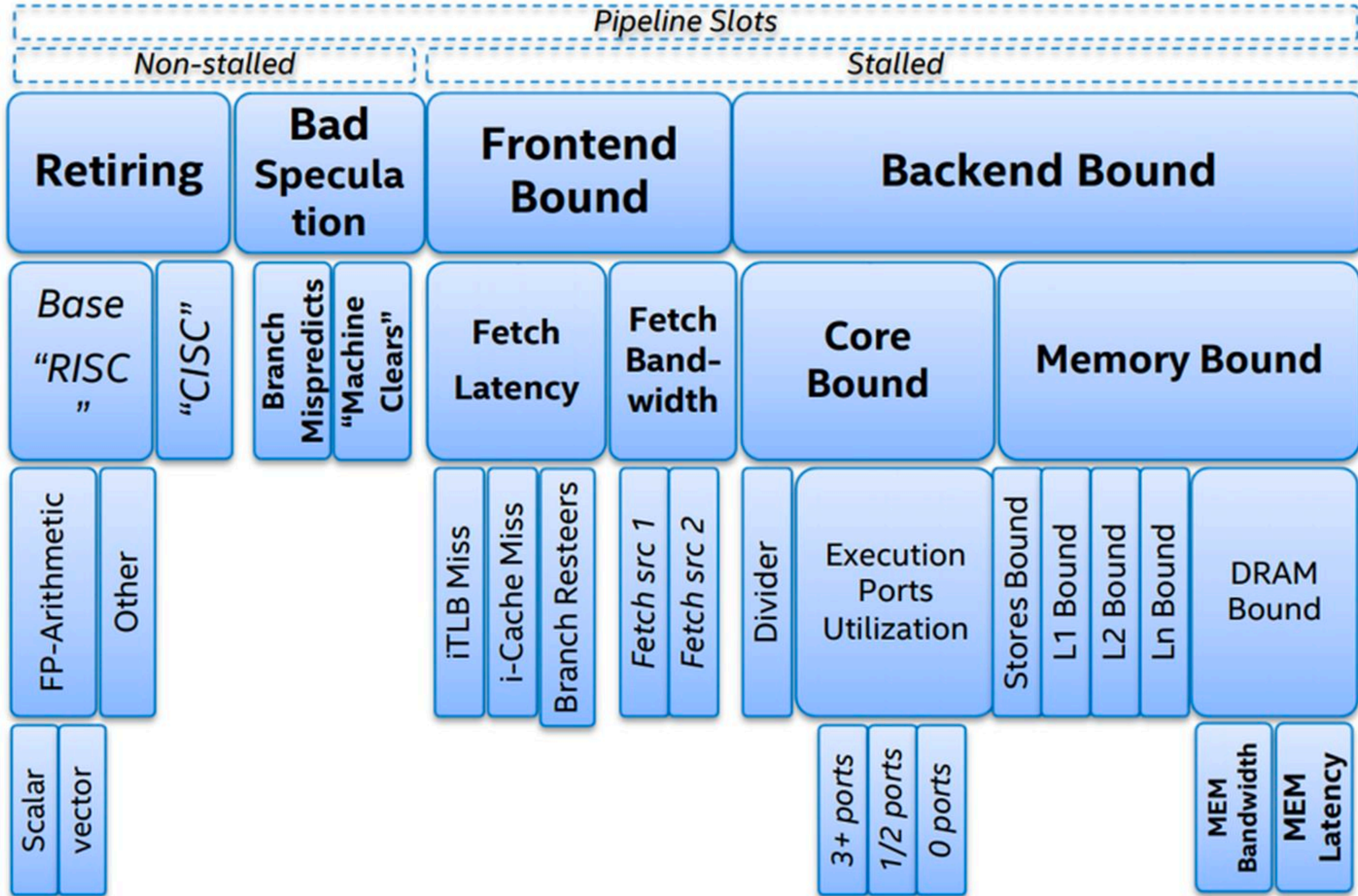
- Multi-level feedback ready queue
- Block queue
- Multiple execution threads
- One Poller thread



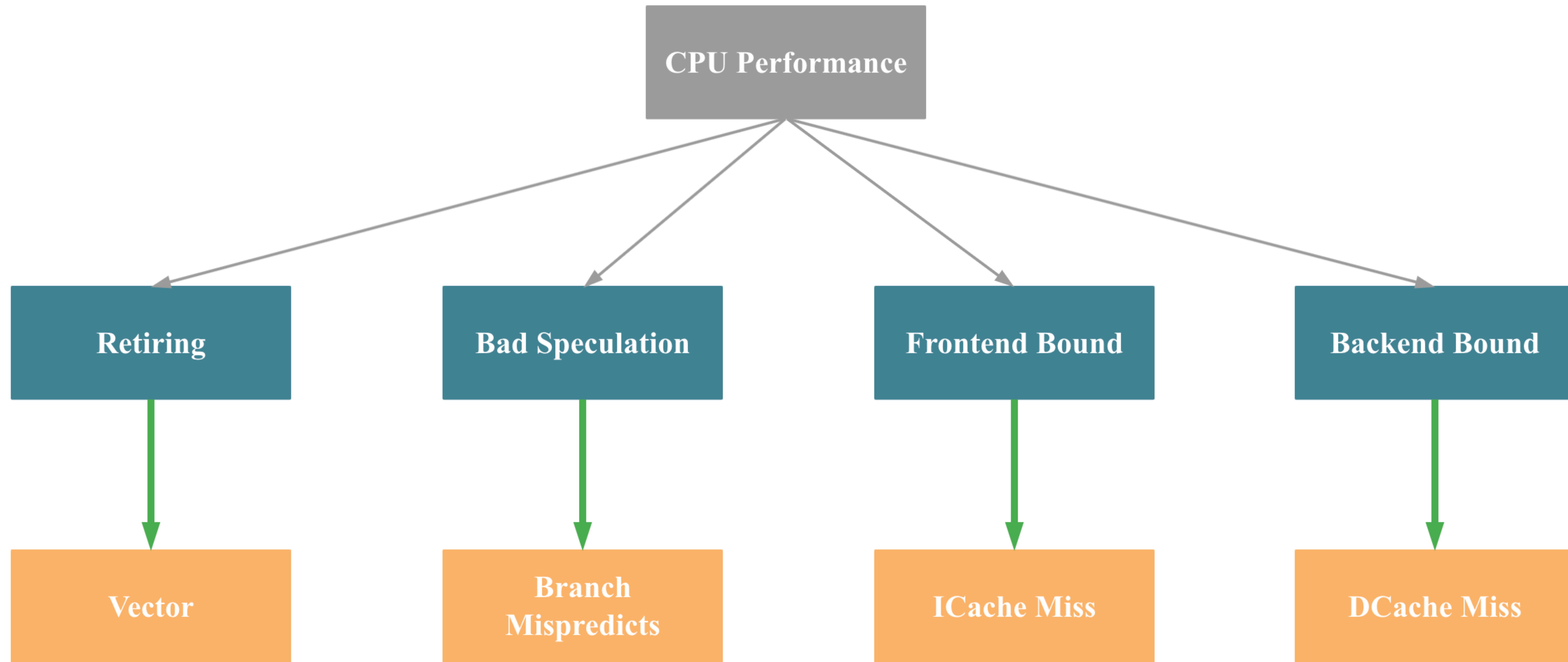
向量化执行：CPU Time

CPU Time = Instruction Number * CPI * Clock Cycle Time

CPU Performance Analysis Top-Down Hierarchy



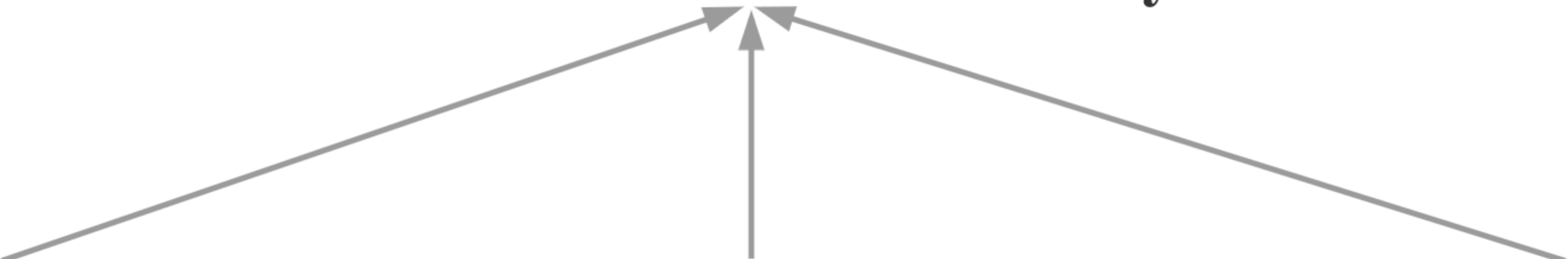
CPU Performance Analysis Top-Down Hierarchy



CPU Time

Vector

CPU Time = Instruction Number * CPI * Clock Cycle Time

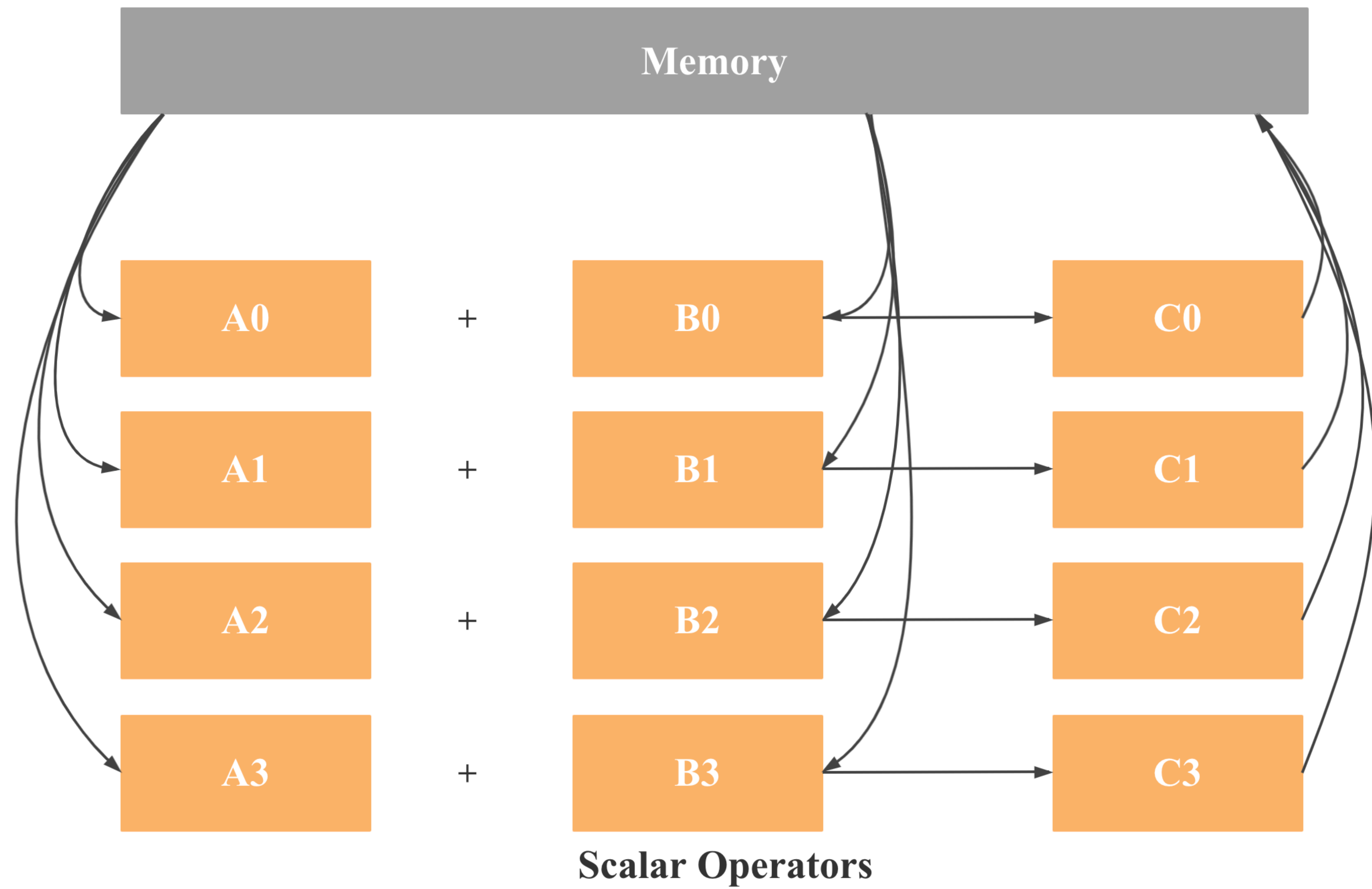


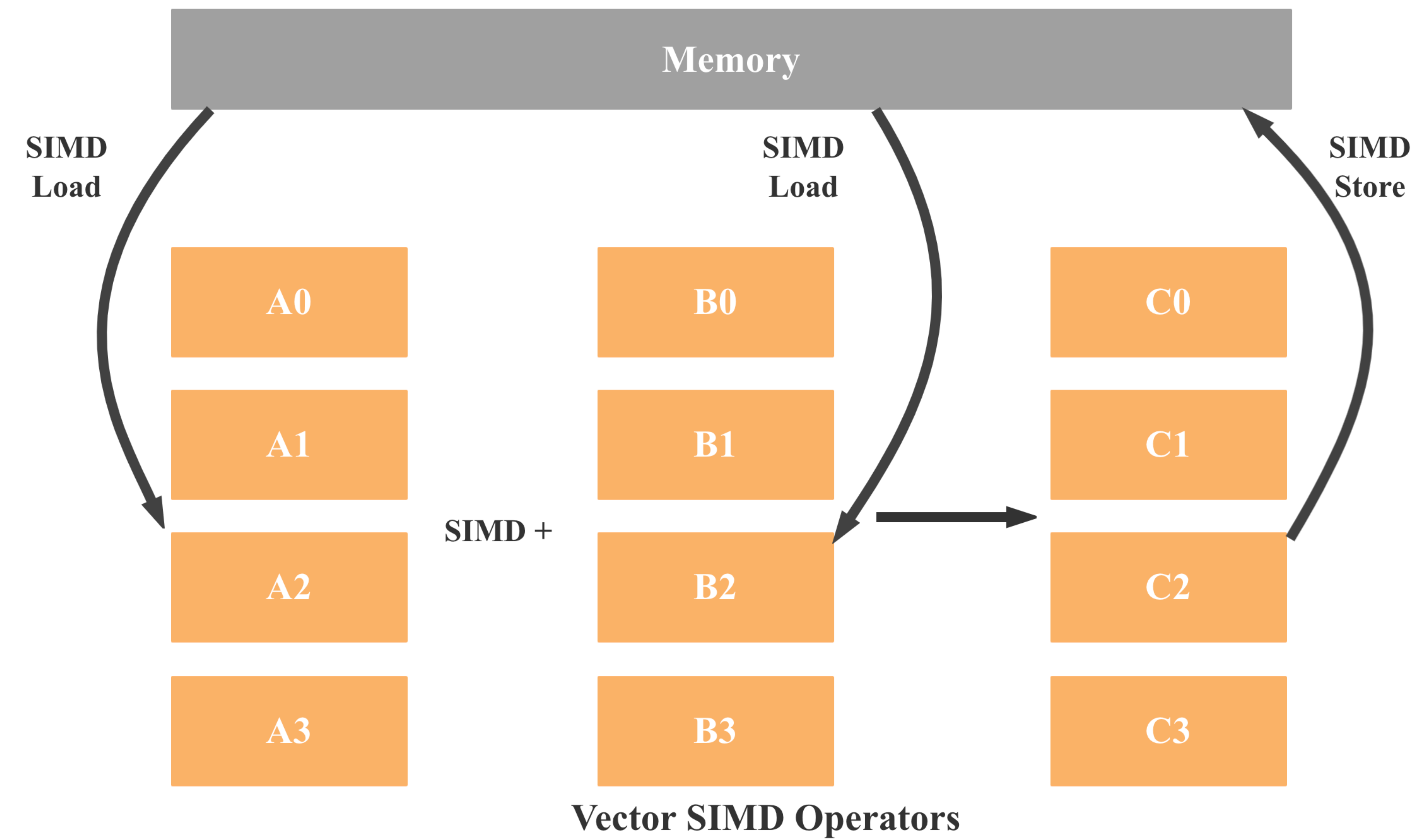
Branch Mispredicts

ICache Miss

DCache Miss

What SIMD



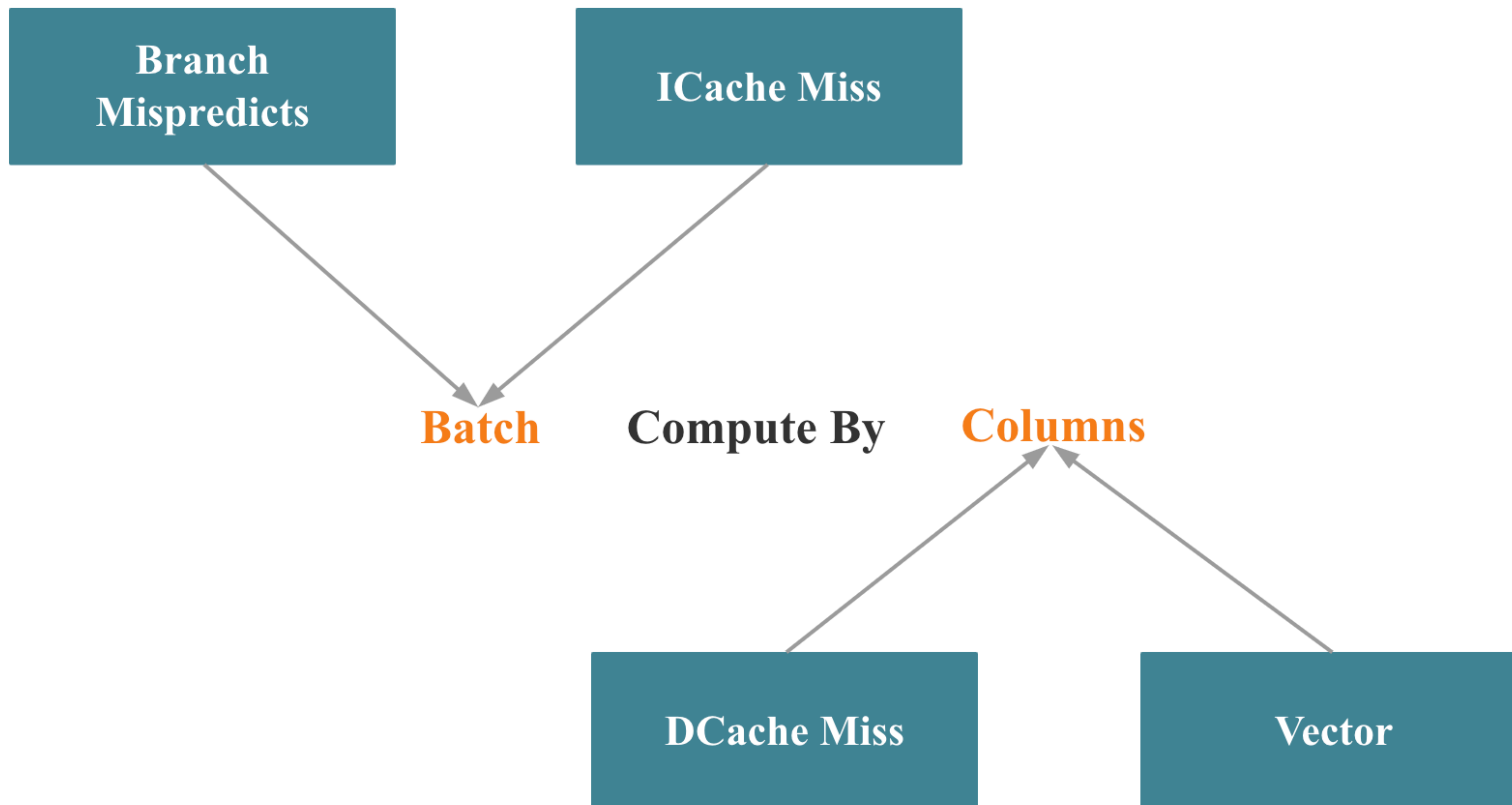


SISD

SIMD

什么是向量化执行

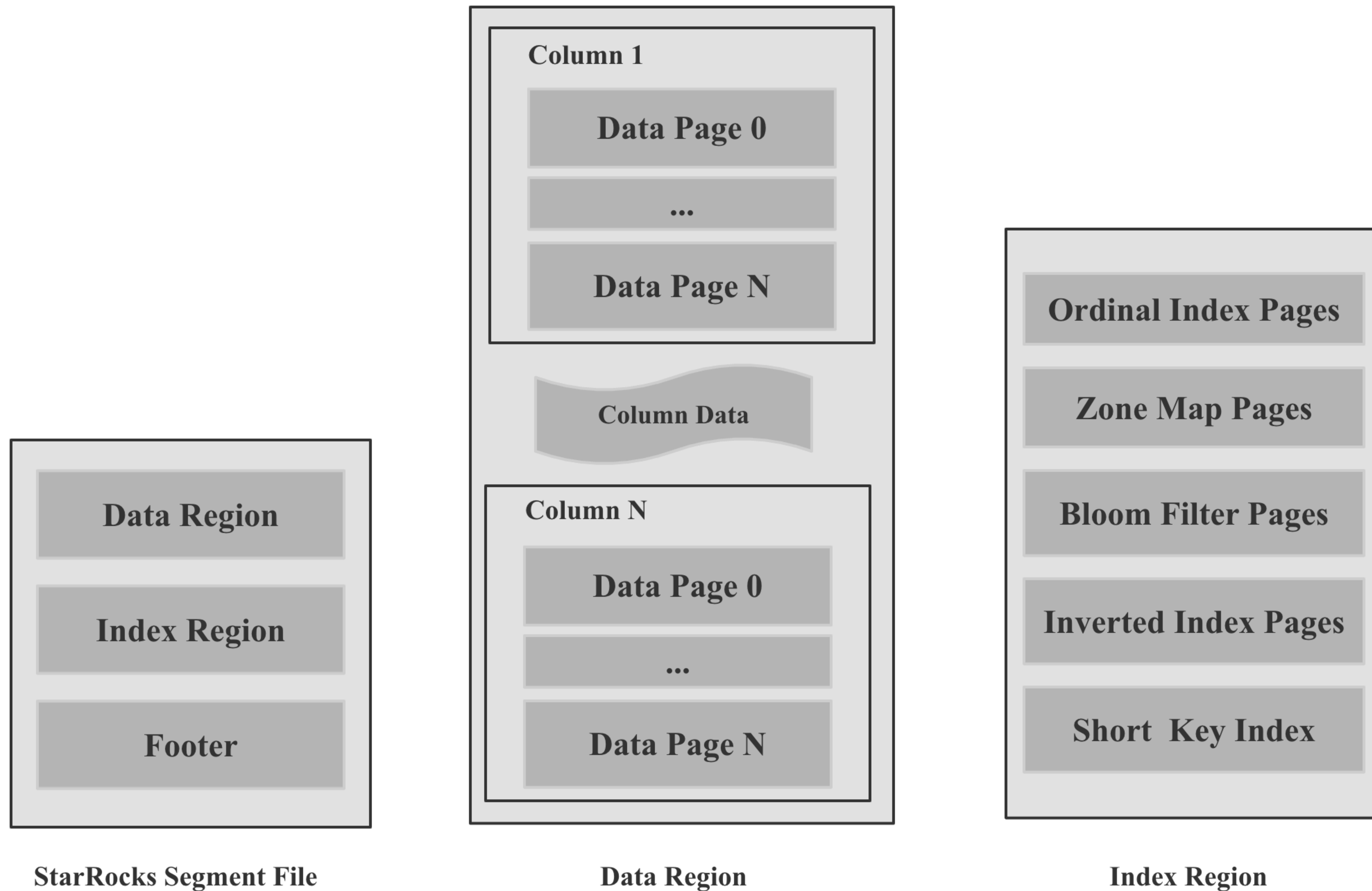
- 表达式
- 算子



数据库向量化执行的关键点

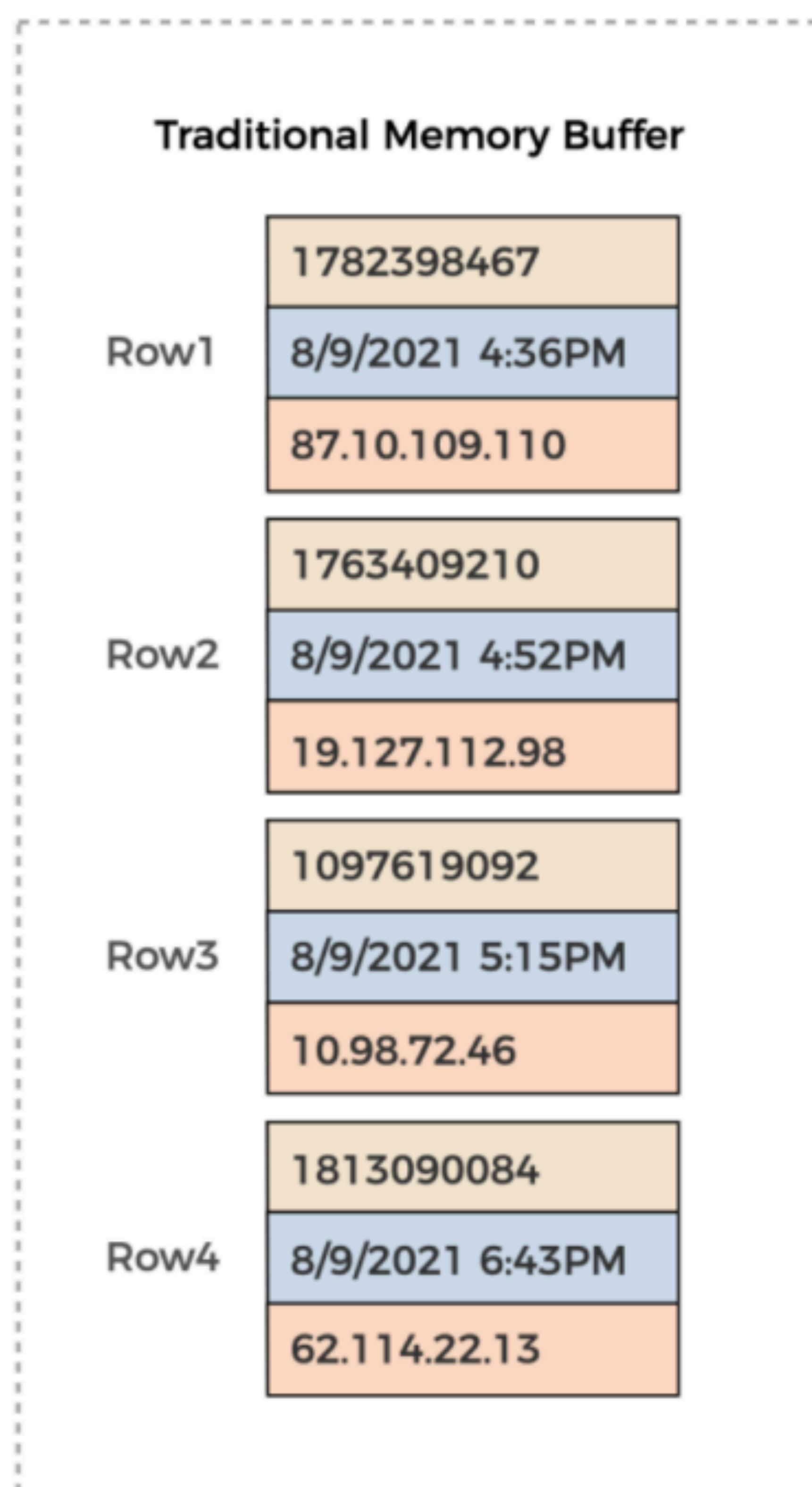
- Data Use **Column Layout** All Time: Disk, Memory, NetWork
- **All Operations** Need To Be Vectorized
- **All Expressions** Need To Be Vectorized
- Use **SIMD** Instructions As Much As Possible
- Redesign **Memory Manage**
- Redesign **Data Structure**
- **Overall Performance** Improve 5x —— All Operations And Expressions Need Improve 5X

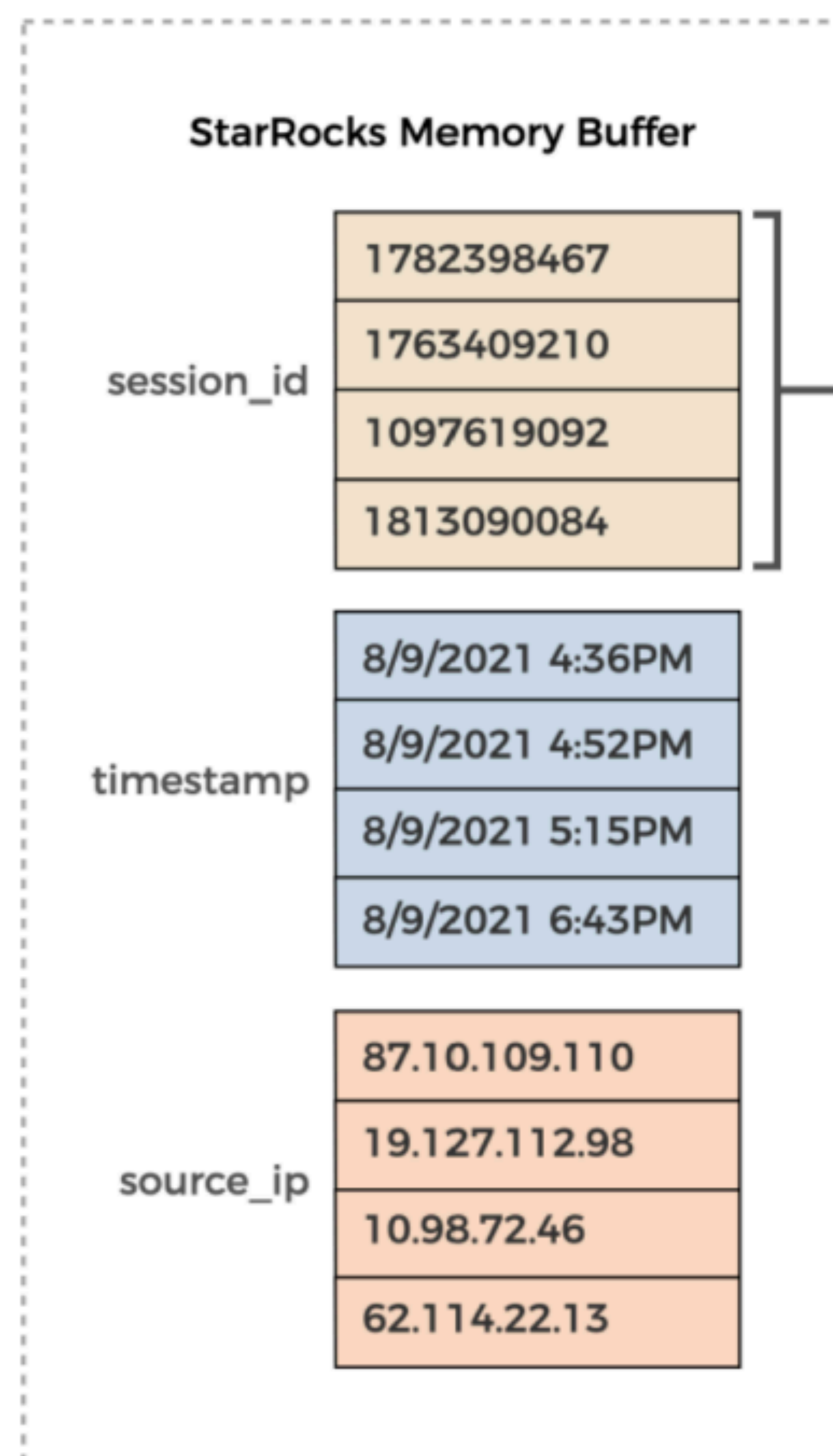
向量化执行 —— Disk Column Store



向量化执行 —— Memory Column Layout

	session_id	timestamp	source_ip
Row1	1782398467	8/9/2021 4:36PM	87.10.109.110
Row2	1763409210	8/9/2021 4:52PM	19.127.112.98
Row3	1097619092	8/9/2021 5:15PM	10.98.72.46
Row4	1813090084	8/9/2021 6:43PM	62.114.22.13





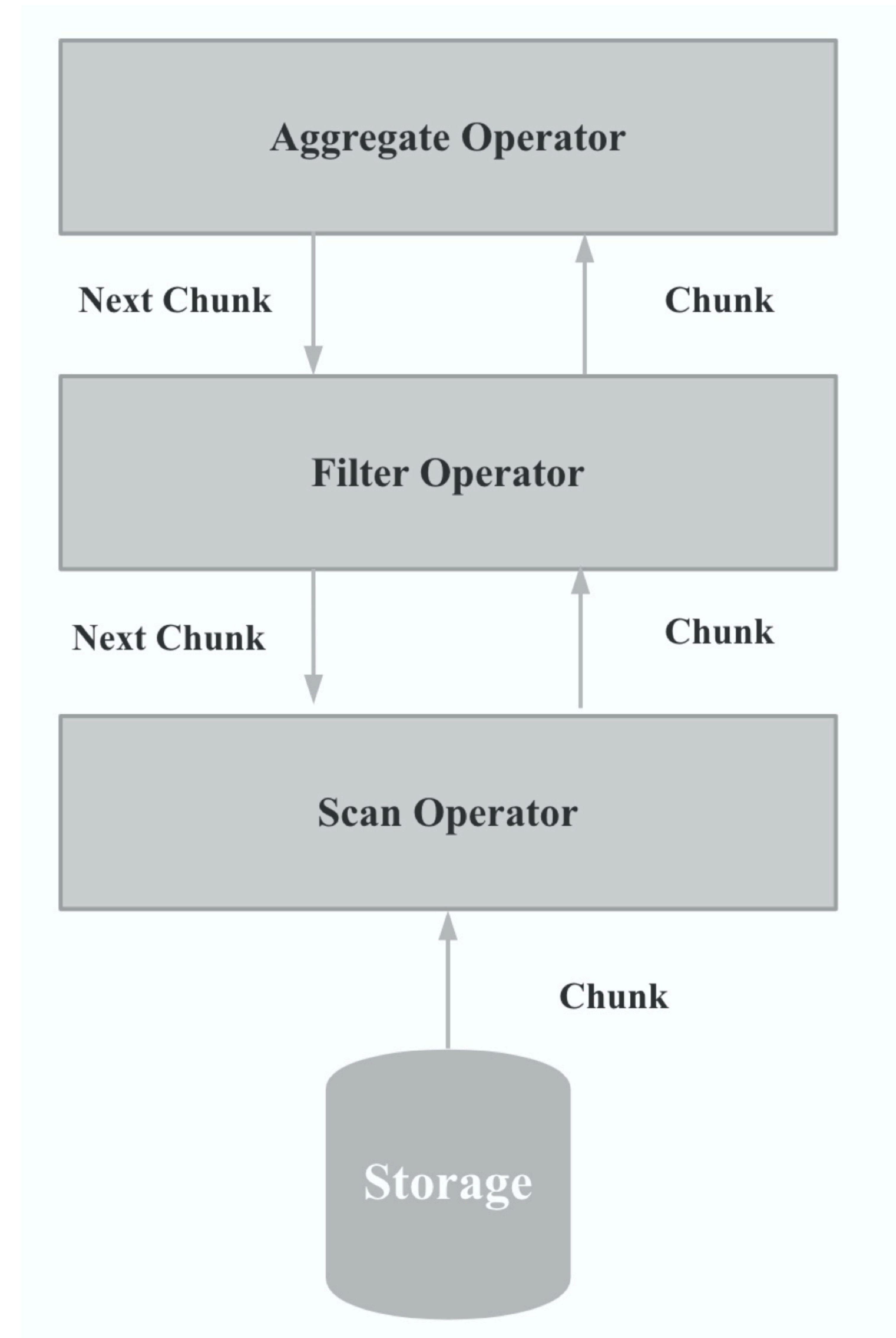

```
SELECT * FROM clickstream
WHERE session_id = 1782398467
```

向量化执行 —— 算子向量化

批量 按列 执行

- 更少的虚函数调用
- 更少的分支判断
- CPU Cache 更友好
- 易于SIMD优化

按列执行什么时候是 Bad Case?



向量化执行 —— 表达式向量化

Input Null — Output Null

Col 3 = Col 1 + Col 2

Col 1 Null: 0, 0, 0, 1

SIMD Bit Or

Col 2 Null: 0, 0, 1, 0

Col 1 Data: 99, 98, 97, 0

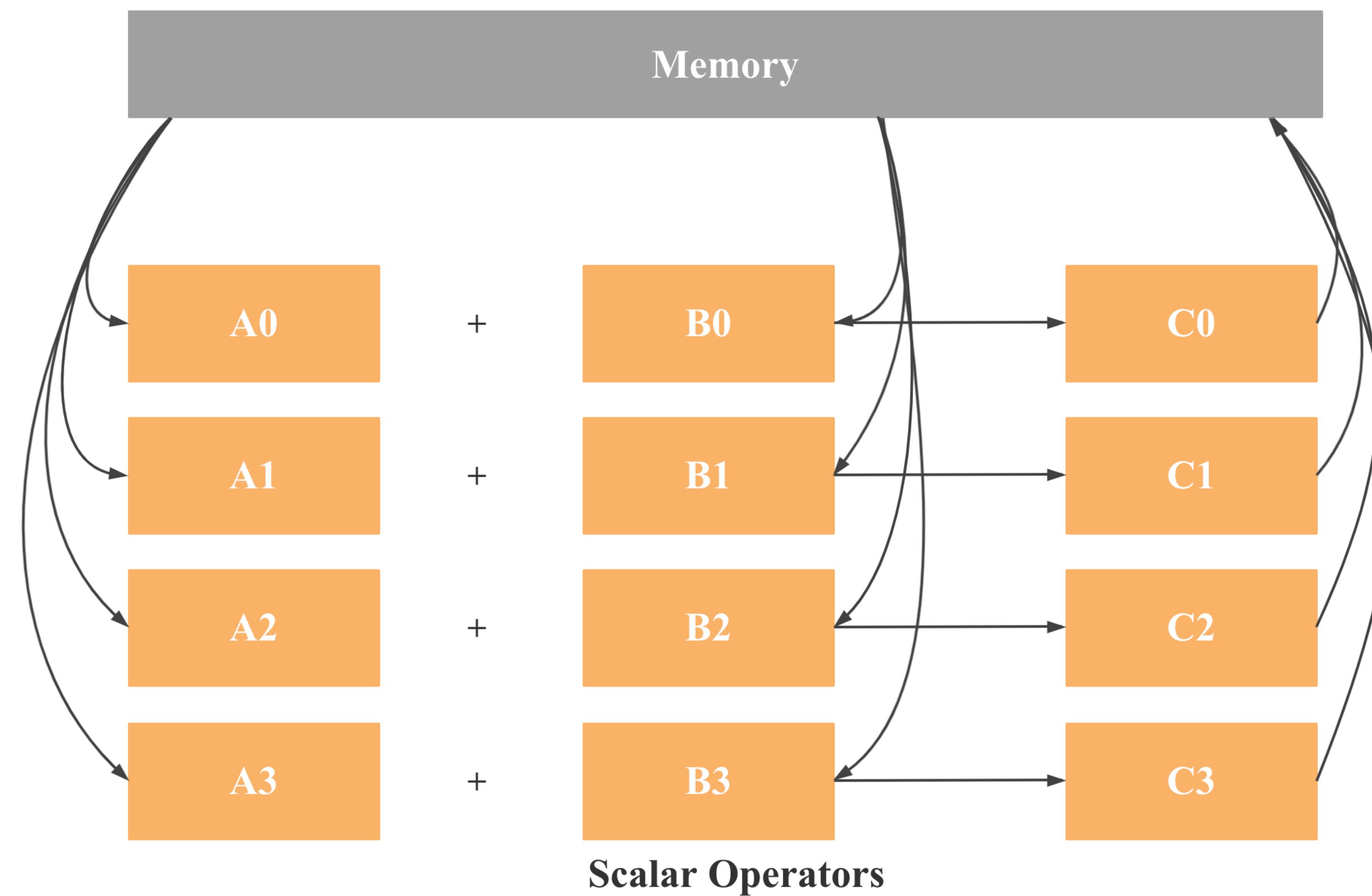
SIMD Add

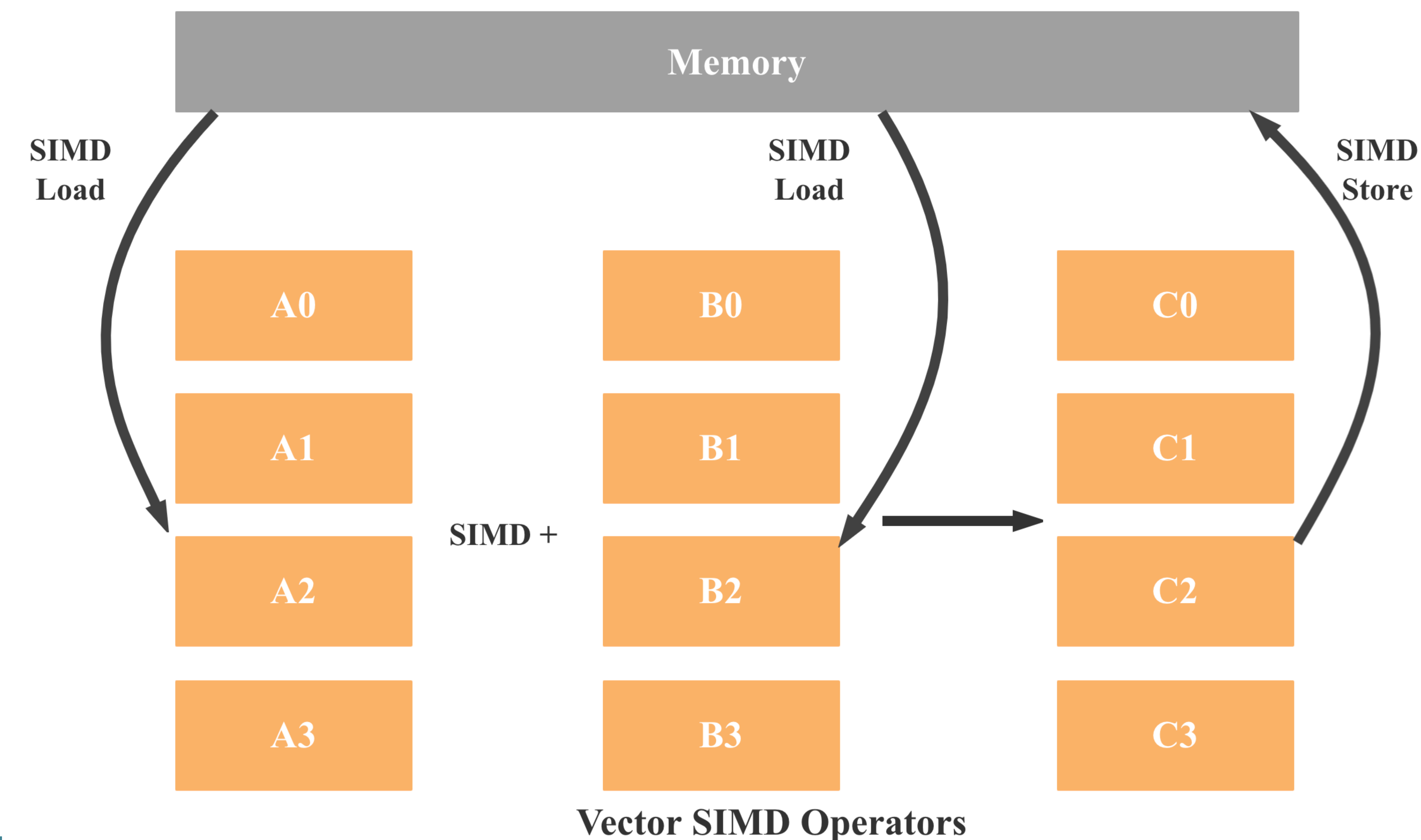
Col 2 Data: 1, 2, 0, 4

Null 列和数量列全部计算什么情况是 Bad Case?

向量化执行 —— 充分的 SIMD 优化

- *SIMD text parsing*
- *SIMD string functions*
- *SIMD json*
- *SIMD join runtime filter*
- *SIMD case when*
- *SIMD gather*
- *SIMD memcpy*
- *SIMD memcmp*
- *SIMD branchless*
- *SIMD filter*
- *SIMD aggregate*
- *SIMD join*





向量化执行：Ascii Substring SIMD 优化 Case

```
char tail_has_error = 0;
for (size_t i = 0; i < len; i++) {
    tail_has_error |= src[i];
}
return !(tail_has_error & 0x80);
```

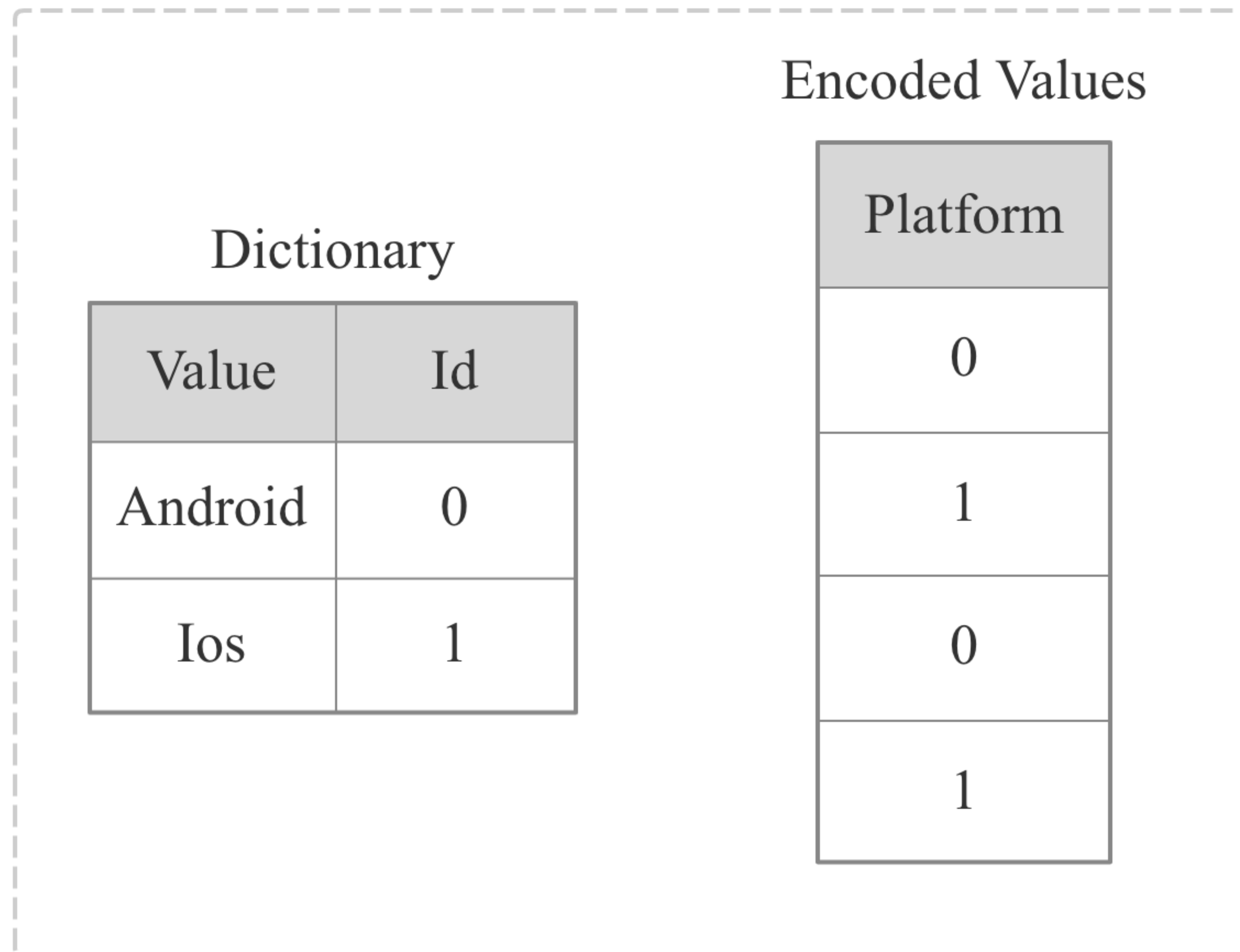
Validate Ascii String

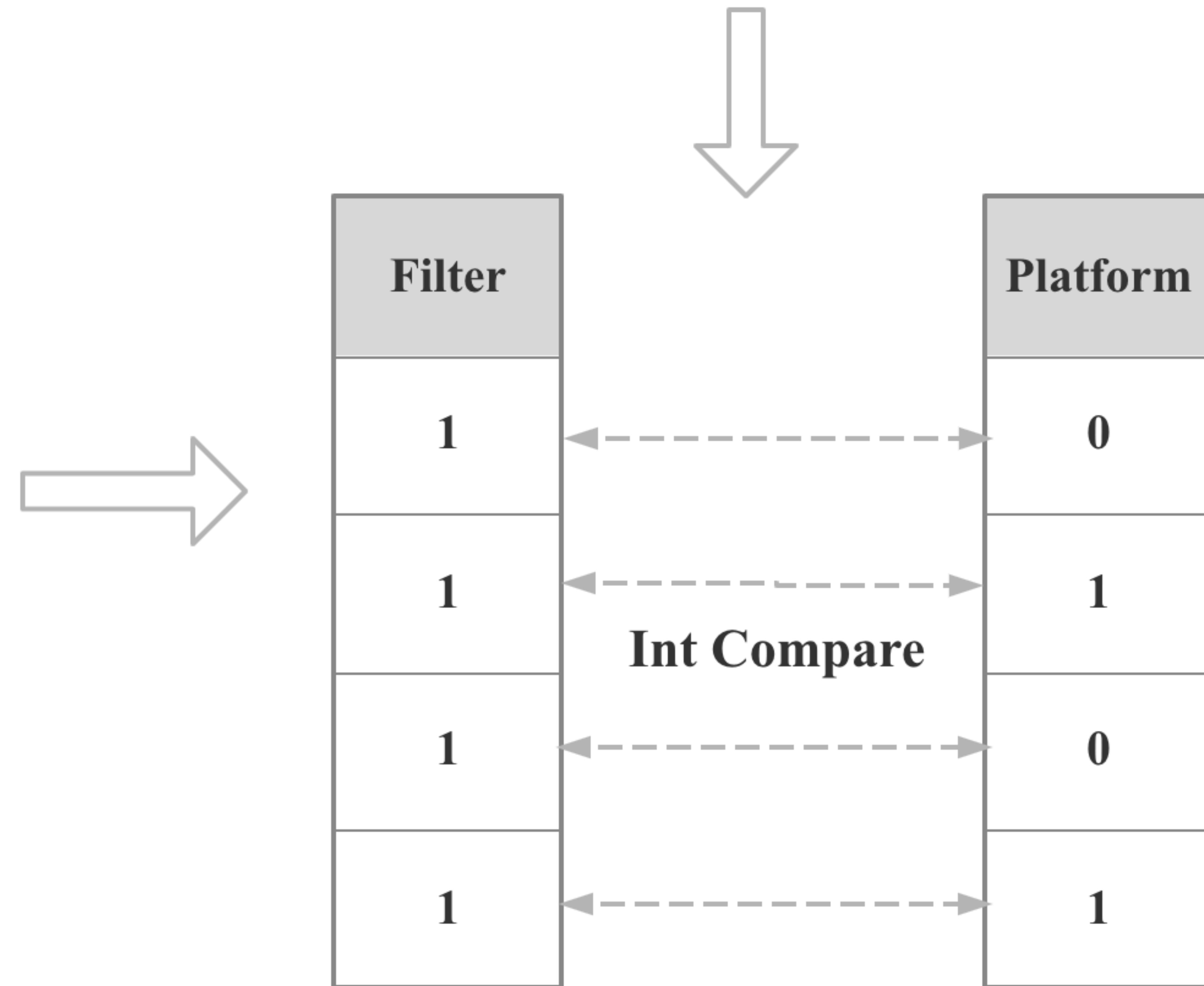
Ascii Chars From 0x00 To 0x7F

```
__m256i has_error = _mm256_setzero_si256();
if (len >= 32) {
    for (size_t i = 0; i <= len - 32; i += 32) {
        __m256i current_bytes = _mm256_loadu_si256((const __m256i*
        has_error = _mm256_or_si256(has_error, current_bytes);
    }
}
int error_mask = _mm256_movemask_epi8(has_error);
```

5X Performance Improvement

向量化执行 —— 低基数字典优化

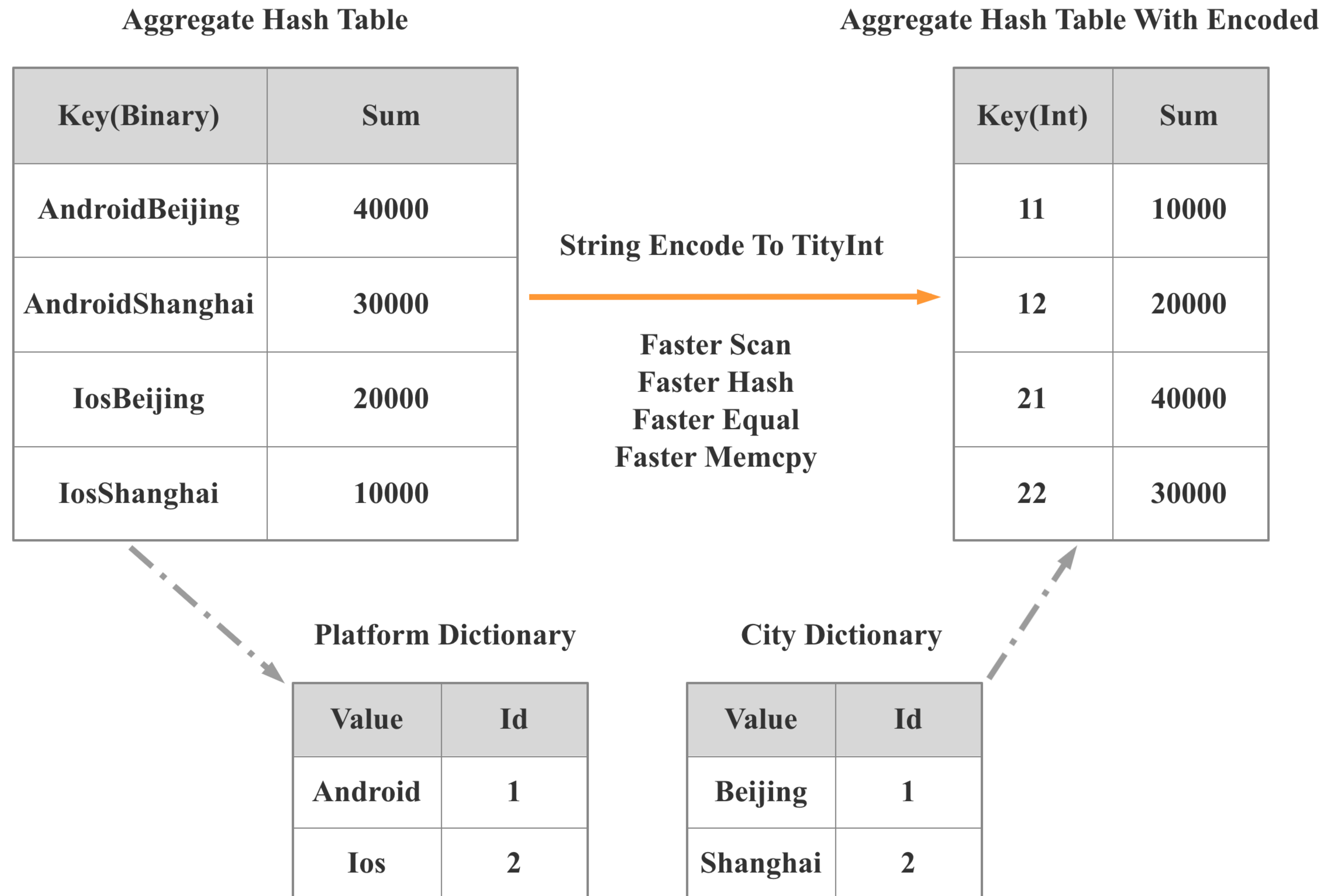




Int Compare is Very Faster Than String

向量化执行 —— 全局低基数字典优化

- Scan
- Filter
- Agg
- Sort
- Join
- String Functions



向量化执行 —— HLL Memory Manage

- Allocate By Chunk
- Reuse Memory

```
+ HyperLogLog::~HyperLogLog() {
+     if (_registers.data != nullptr) {
+         ChunkAllocator::instance()->free(_registers);
+     }
+ }
+
  // Convert explicit values to register format, and clear explicit values.
  // NOTE: this function won't modify _type.
  void HyperLogLog::_convert_explicit_to_register() {
      DCHECK(_type == HLL_DATA_EXPLICIT)
              << "_type(" << _type << ") should be explicit(" << HLL_DATA_EXPLICIT << ")";
-     _registers.clear();
-     _registers.resize(HLL_REGISTERS_COUNT, 0);
+     DCHECK_EQ(_registers.data, nullptr);
+     ChunkAllocator::instance()->allocate(HLL_REGISTERS_COUNT, &_registers);
+     memset(_registers.data, 0, HLL_REGISTERS_COUNT);
+
```

5X Performance Improvement

向量化执行 —— CPU Cache 优化

StarRocks

Table 2.2 Example Time Scale of System Latencies

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50–150 μs	2–6 days
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–317 years
OS virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware (HW) virtualization system reboot	40 s	4 millennia
Physical system reboot	5 m	32 millennia

With SIMD, the performance bottleneck is from CPU bound to memory bound

向量化执行 —— CPU Cache 优化

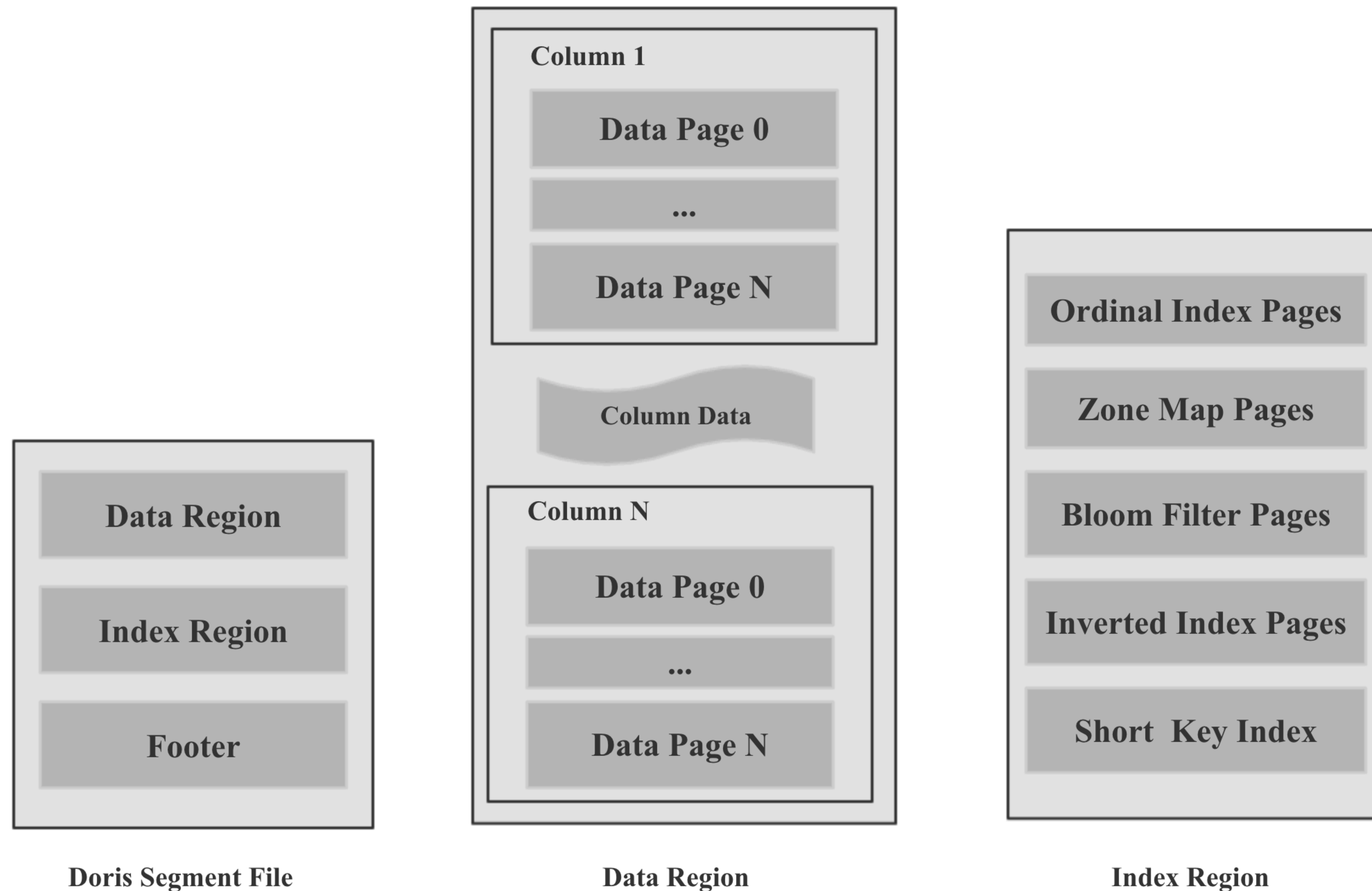
The Performance Bottleneck Will Vary

向量化执行 —— CPU Cache 优化

- Improve Locality (Spatial And Temporal)
- Align The Code And Data
- Reduce Memory Footprint
- Block
- Prefetch

存储：Column Store

- 高效IO
- 高效编码压缩
- 丰富索引加速过滤

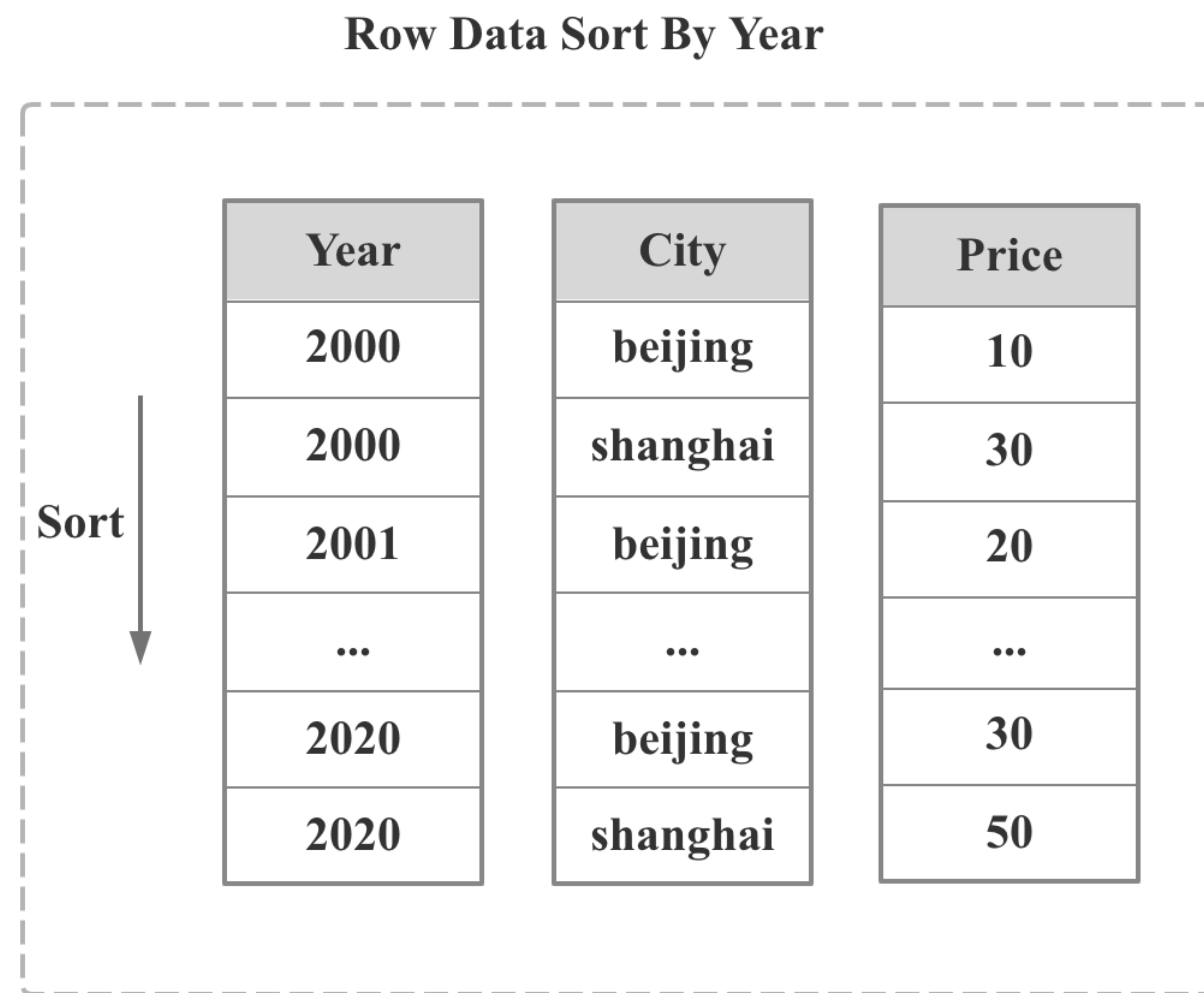


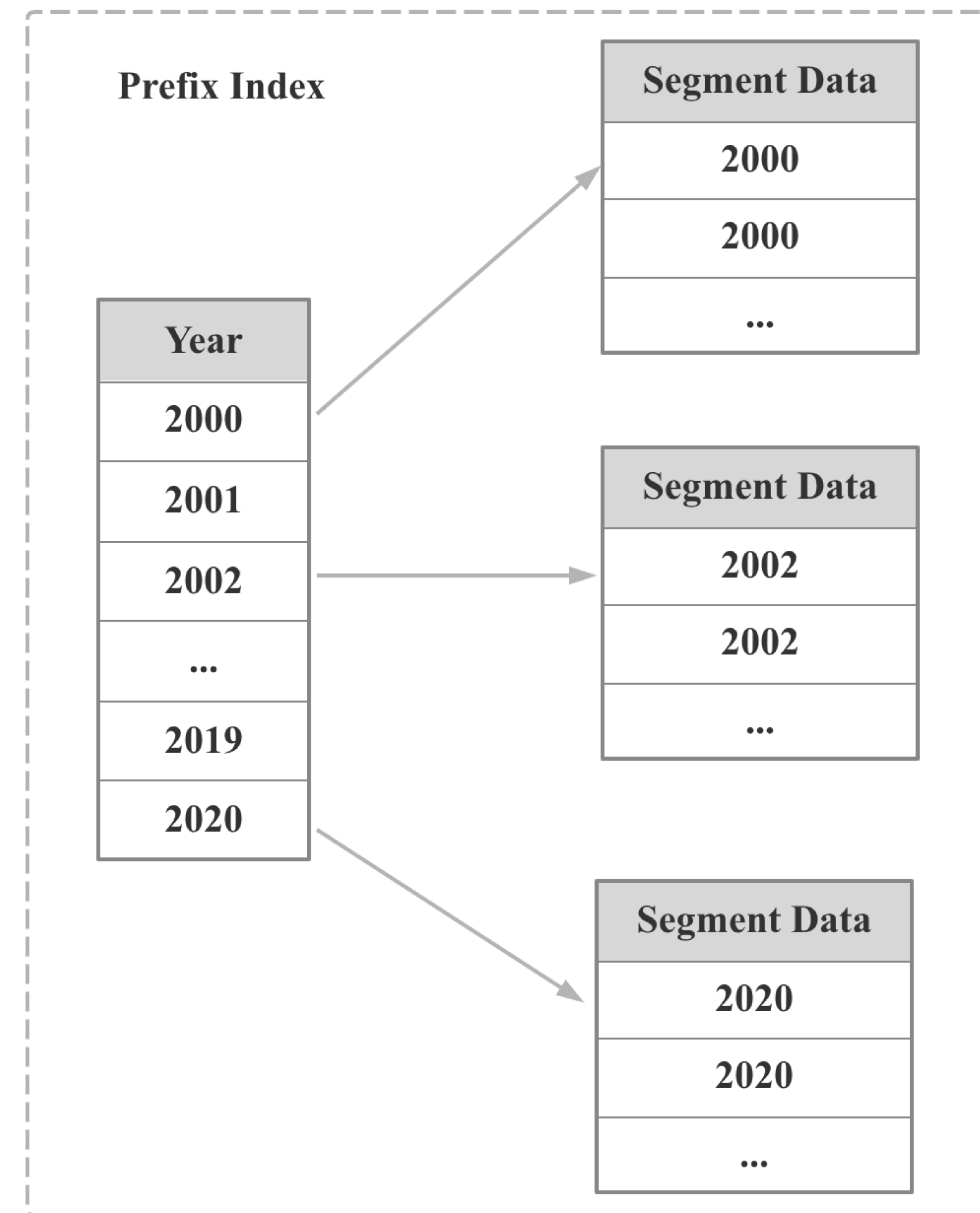
存储：前缀索引

•基于排序列

•索引粒度：1024行

•索引Key: 36 字节





显著加速前缀Key列过滤场景

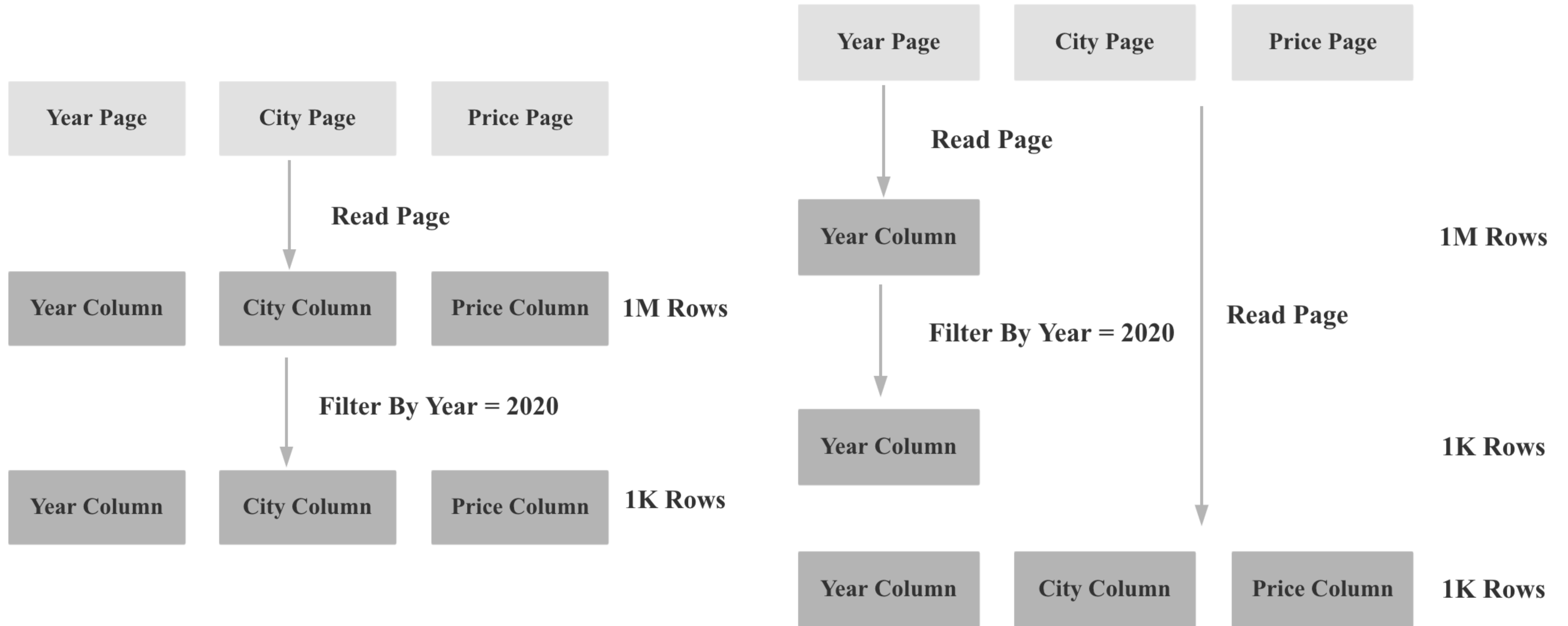
存储：Bitmap 倒排索引

显著加速后缀过滤和多列过滤场景

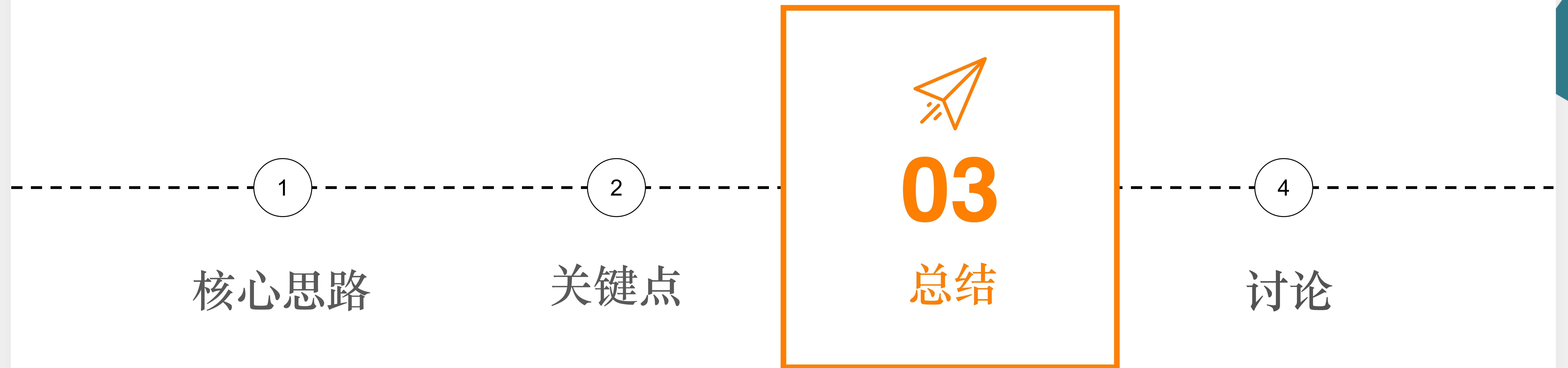
Bitmap Index 原理

存储：延迟物化

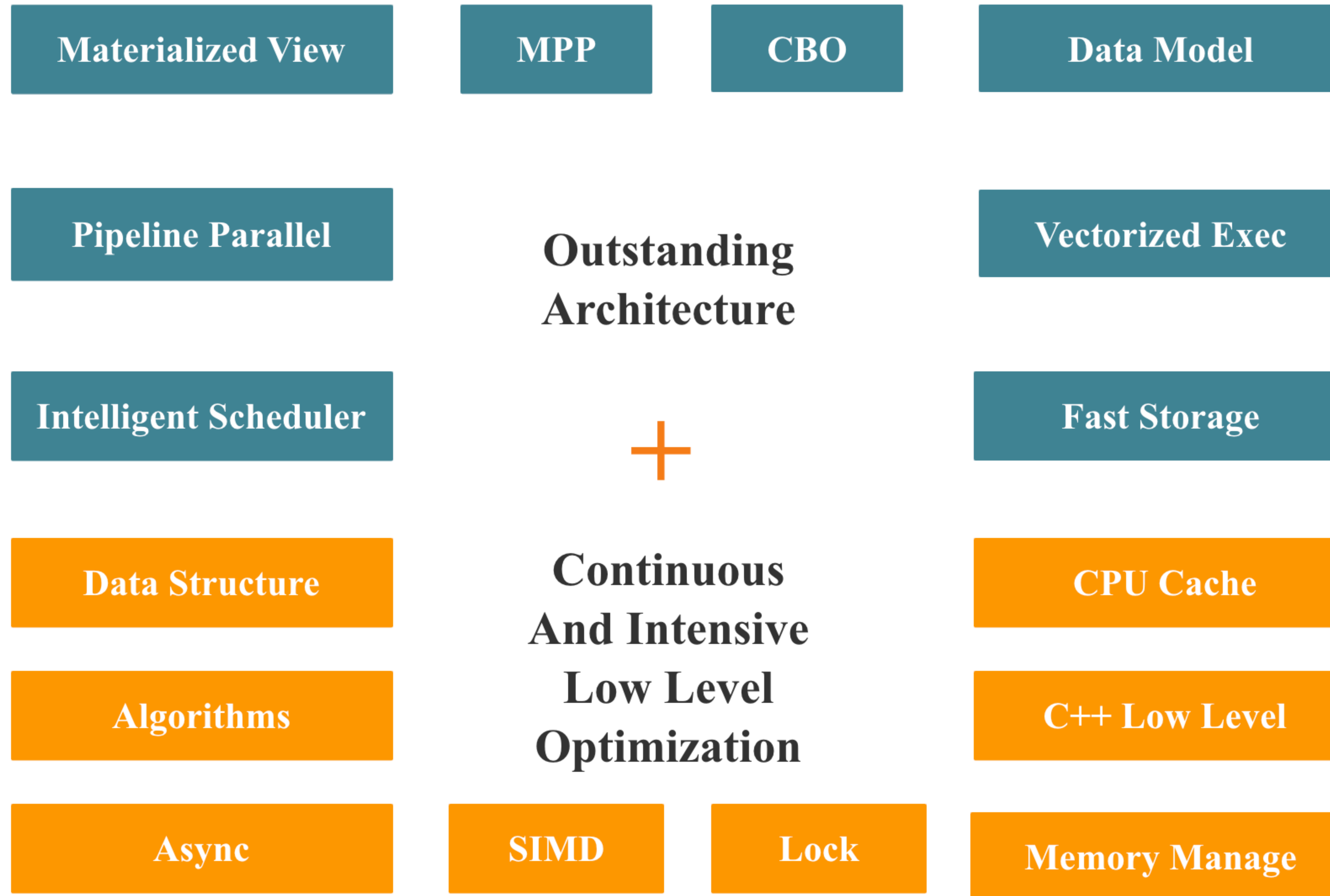
Select Year, City, Price From Table Where Year = 2020



如何打造一款极速数据库

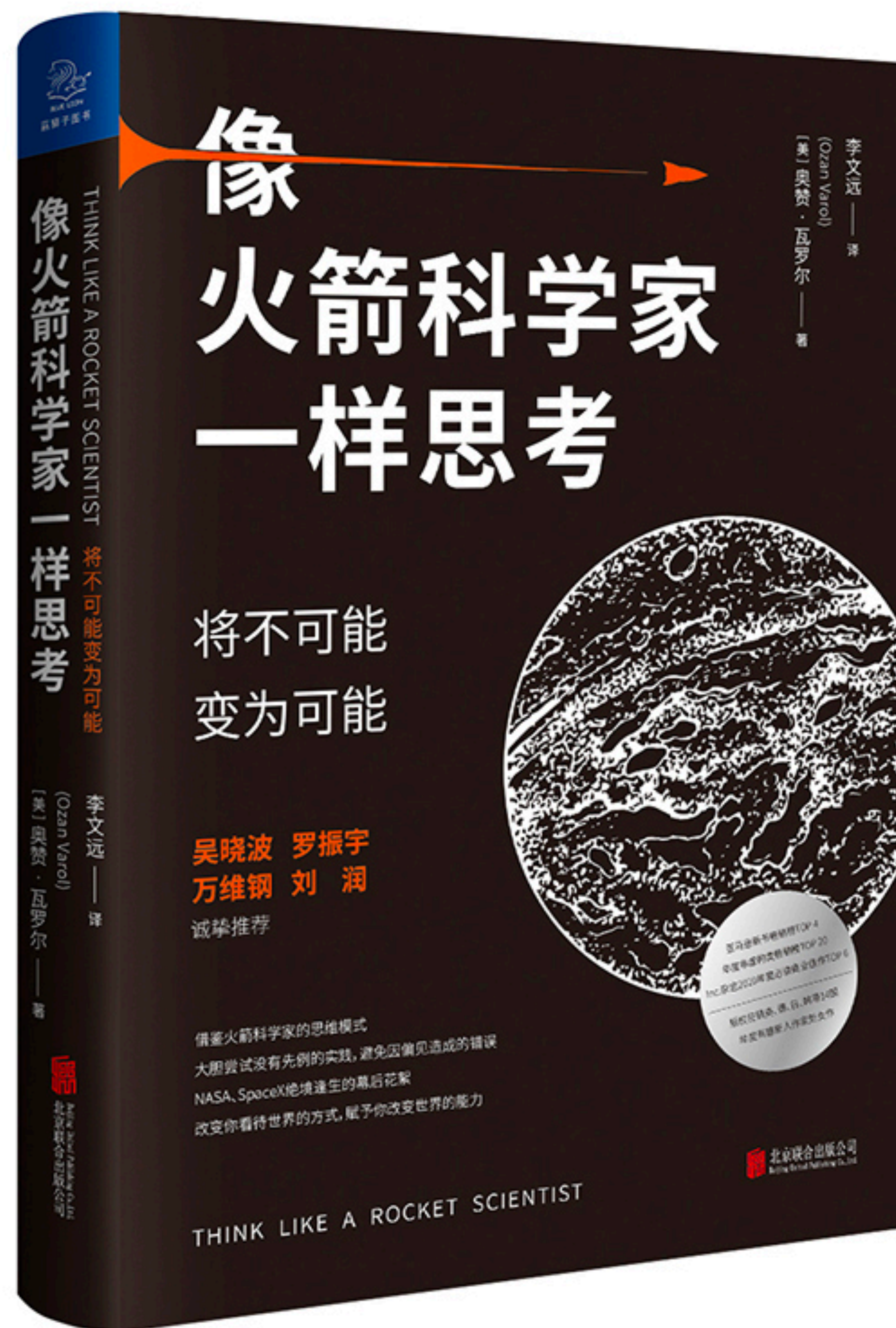


High Performance Database Need



推荐 —— To Be Impossible

- 探险思维
- 第一性原理
- 奇迹思维
- 解决者思维
- 证伪思维
- 压力测试
- 迭代思维



学习资料

- 数据库学习资料
- How To Build A Fast DataBase
- 《OLAP 数据库性能优化指南》

2023

Thanks