Tencent Big Data

MercsDB

Next generation of Tencent OLAP Engine

Tencent TEG LongYue

数据平台部

CONTENTS

Background

- **Data :**

1. Thousand Columns, 10 Billions Rows
2. Index on arbitrary column(s)
3. Real-Time Write
4. Both Row-oriented and Column-oriented
5. Different Indexes

MercsDB

- **Performance :**

1. Second response on 10 Billions rows query
2. MPP: both ad-hoc query and real-time query
3. Real-time Write: 100 Billion rows / day

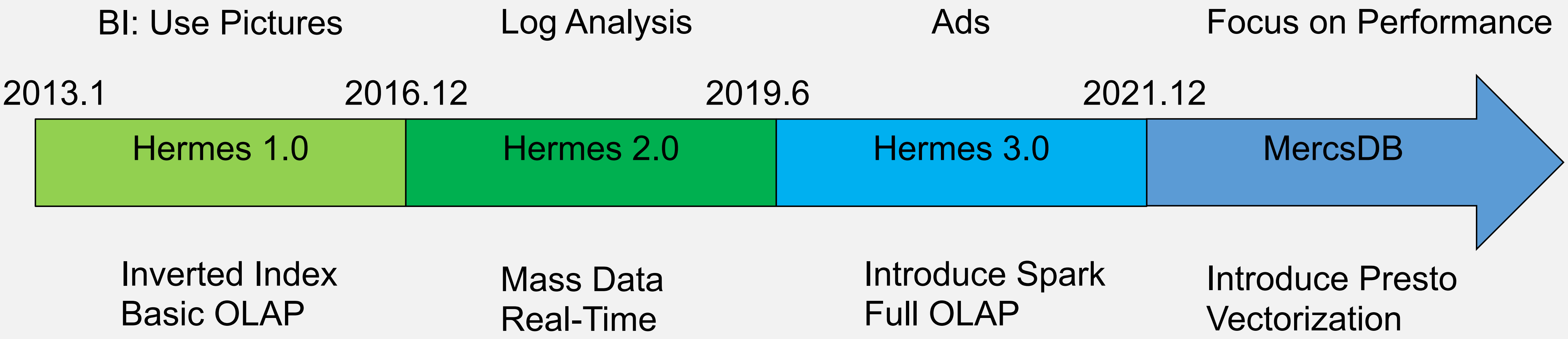
! =

ElasticSearch

Presto/Impala

ClickHouse

History


BI: Use Pictures
Log Analysis
Ads
Focus on Performance
2013.1
2016.12
2019.6
2021.12
Hermes 1.0
Hermes 2.0
Hermes 3.0
MercsDB
Inverted Index
Basic OLAP
Mass Data
Real-Time
Introduce Spark
Full OLAP
Introduce Presto
Vectorization

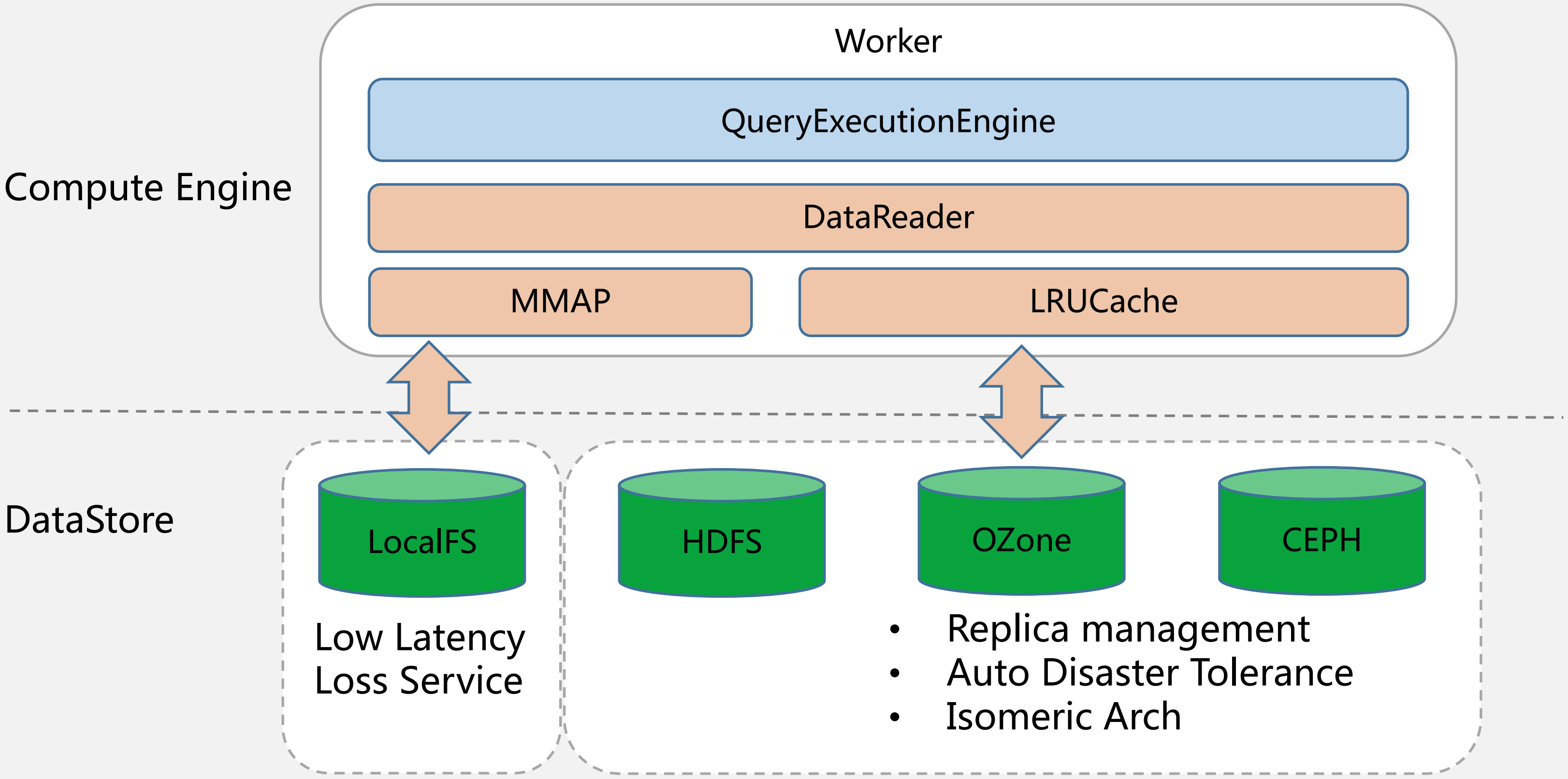
Current Status

Clusters	Query	Storage	Peak IO
5k+ Nodes	**10M / Day**	**Total: 100 PB** **Daily: 1PB**	**100M rows / s**

数据平台部

CONTENTS

Basic Arch



- Districted vs Local
- HA vs Low Latency
- Hot Data vs Cold Data

Column-Oriented & Index

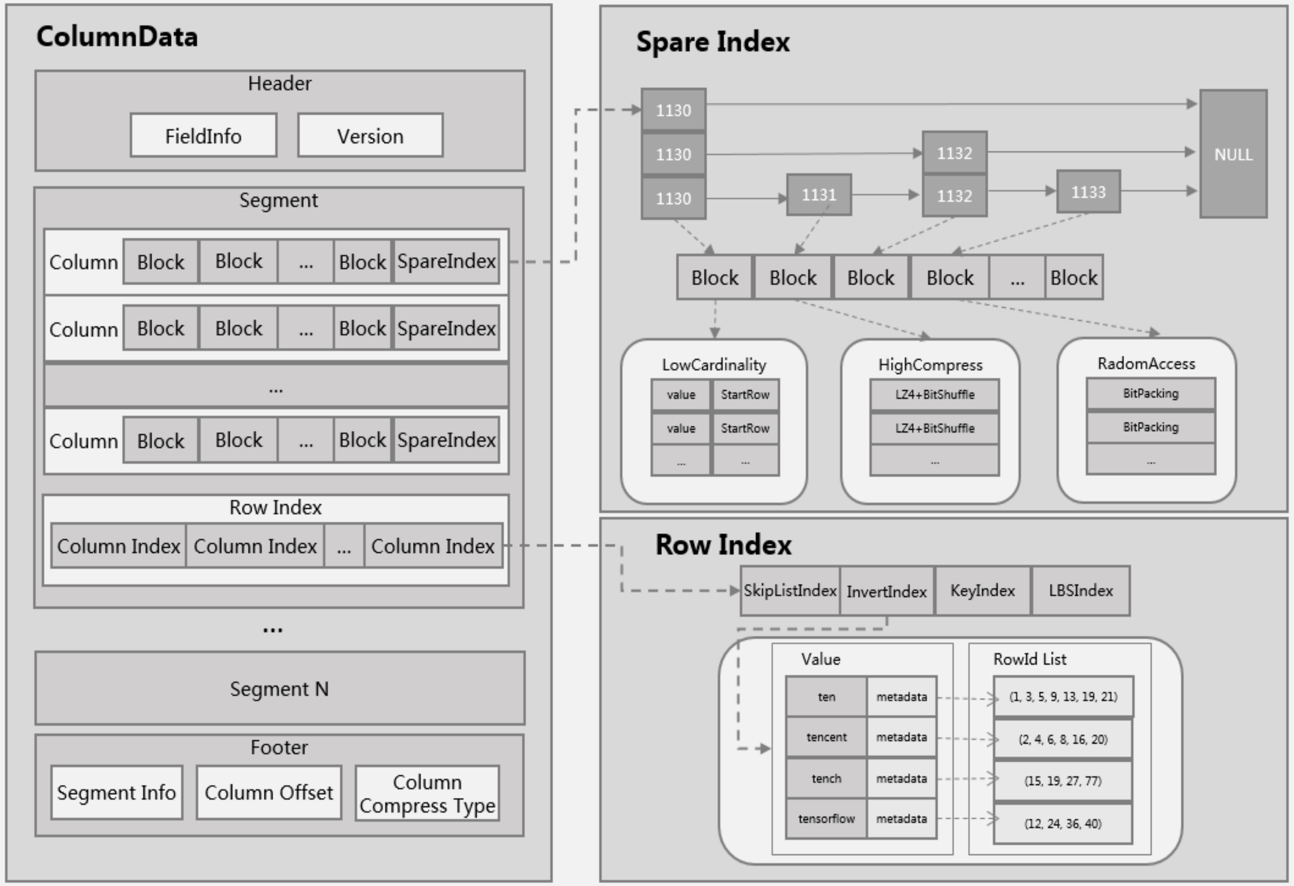
Q1: High QPS?
Q2: Second response for 10Bilion rows?
Q3: Cost-friendly for mass data?

- **Columns:**

1. Retrieval —— Low latency
2. Sorted —— Mass Data
3. Compressed —— Cost-Friendly
4. Nested —— Support parquet

- **Indexes:**

1. SpareIndex
2. SkipListIndex
3. InvertedIndex
4. KeyIndex
5. LBSIndex



Retrieval Column

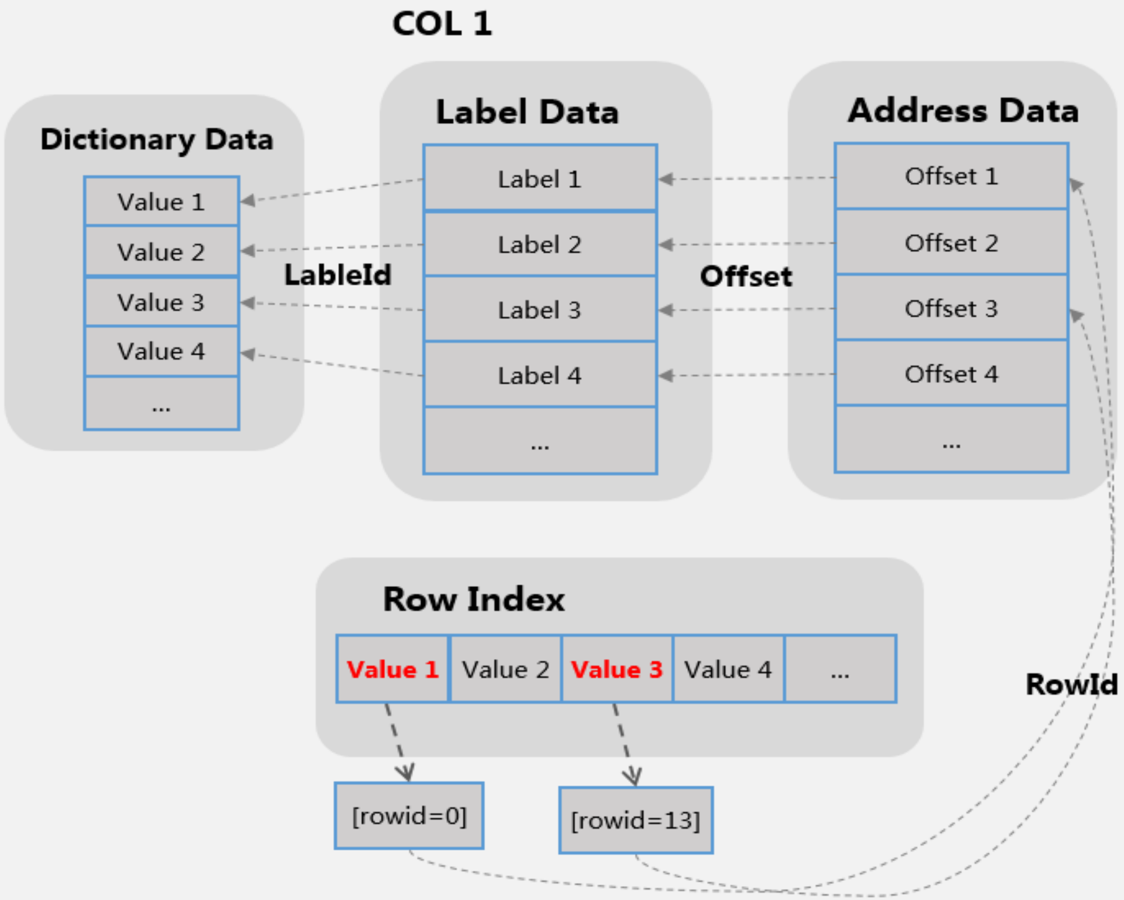
- **Application**

1. Low latency
2. Medium data
3. Simple Query

- **Implementation**

1. Storage -> Time
2. Dictionary Index

Index Size/Origin Data = 40%



Sorted Column

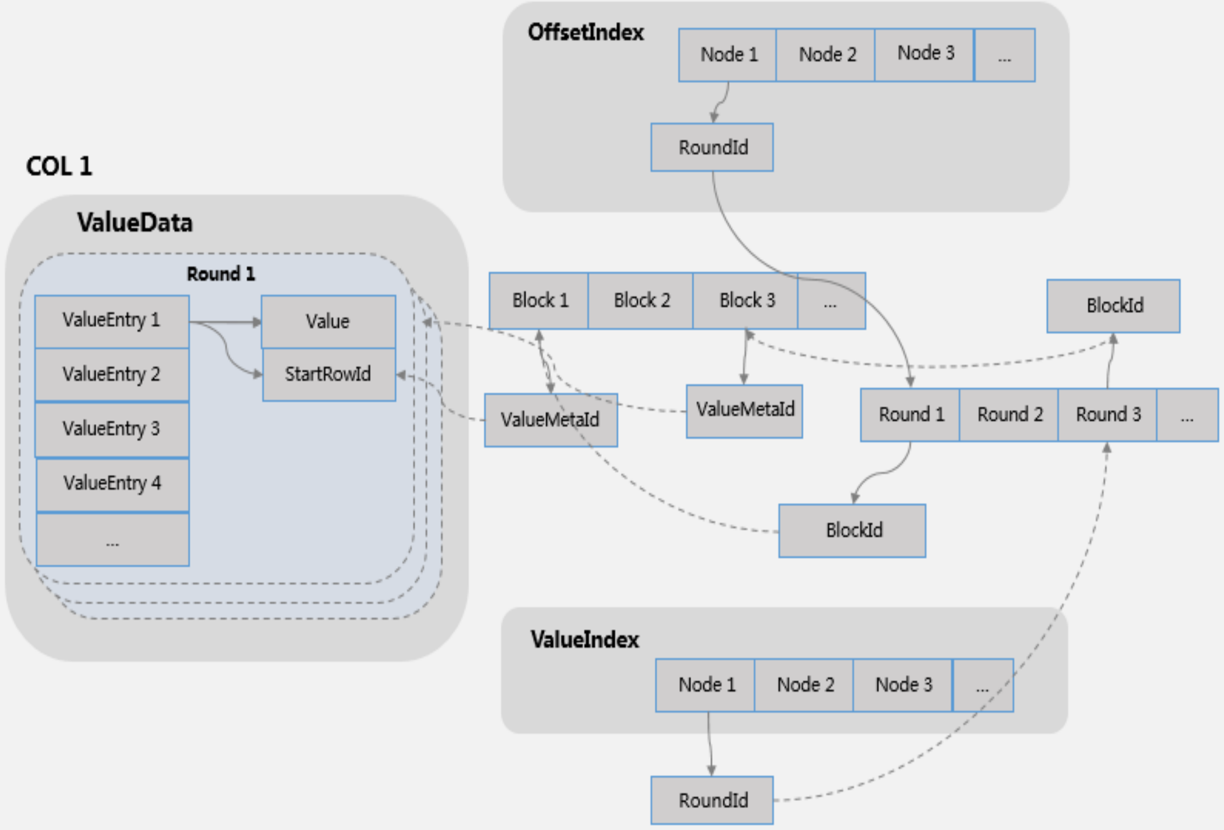
- **Application**

1. Mass Data (much bigger than memory)
2. No other accelerations

- **Implementation**

1. Sorted
2. Index both on data and offset

Improvement: 10x speed up



Compressed Column

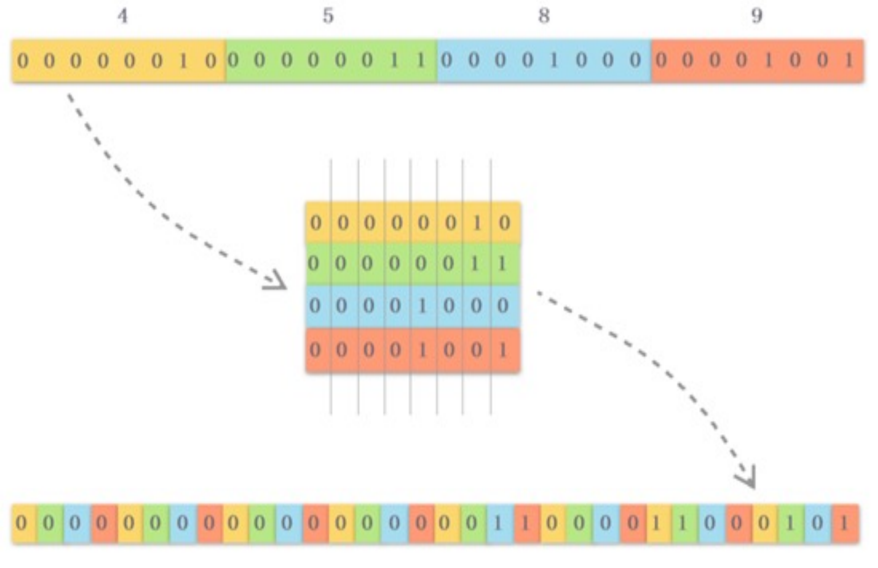
- **Application**

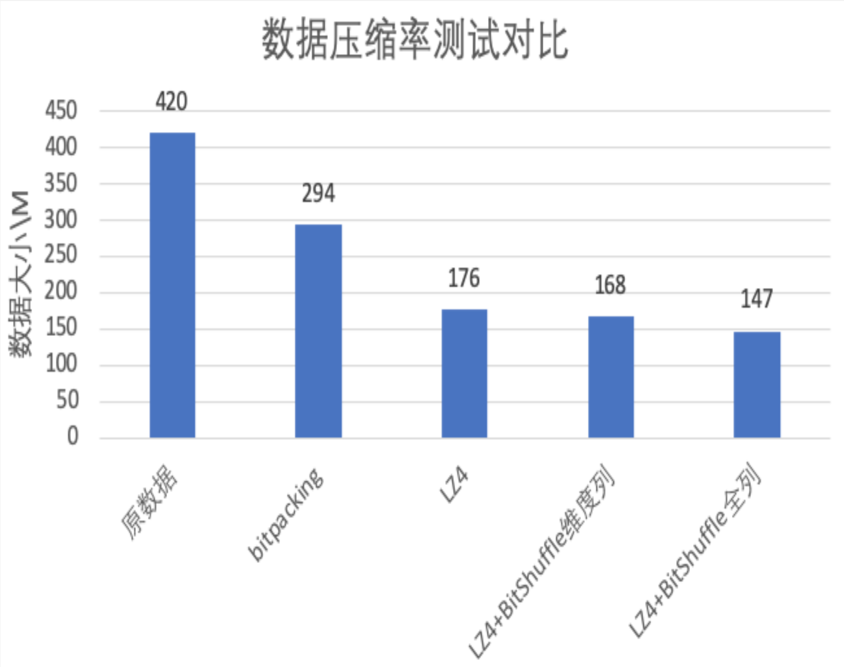
1. High Cardinal
2. Direct compression: low performance

- **Implementation**

1. BitShuffle + LZ4

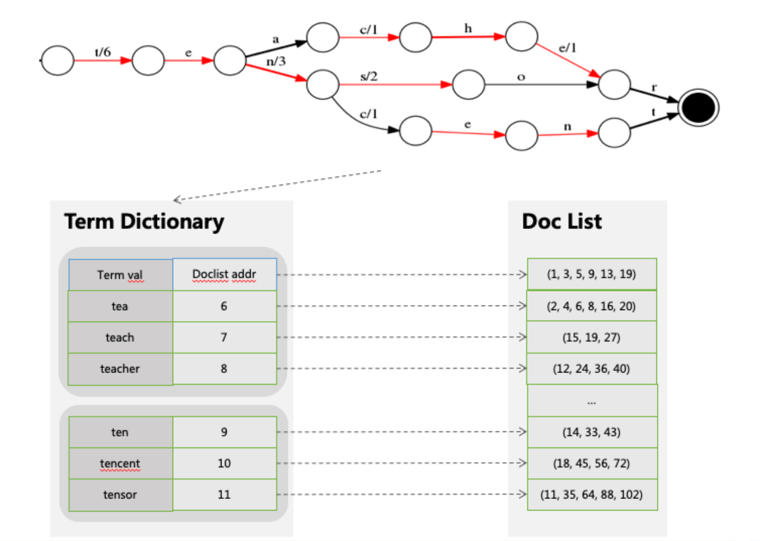
Compression rate: 50% up



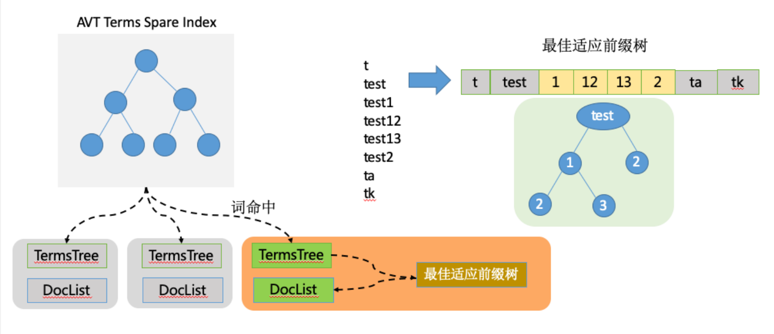


Indexs

Inverted Index

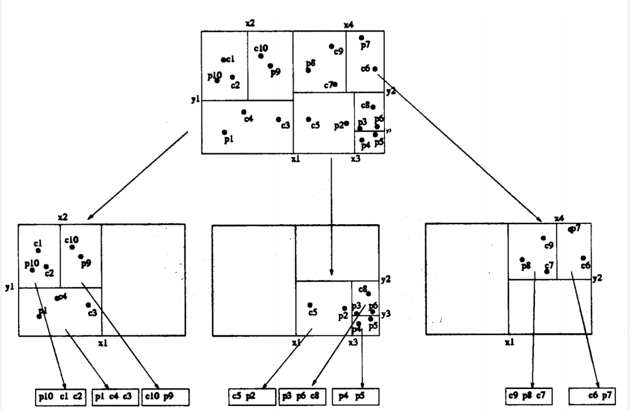


FST KeyIndex



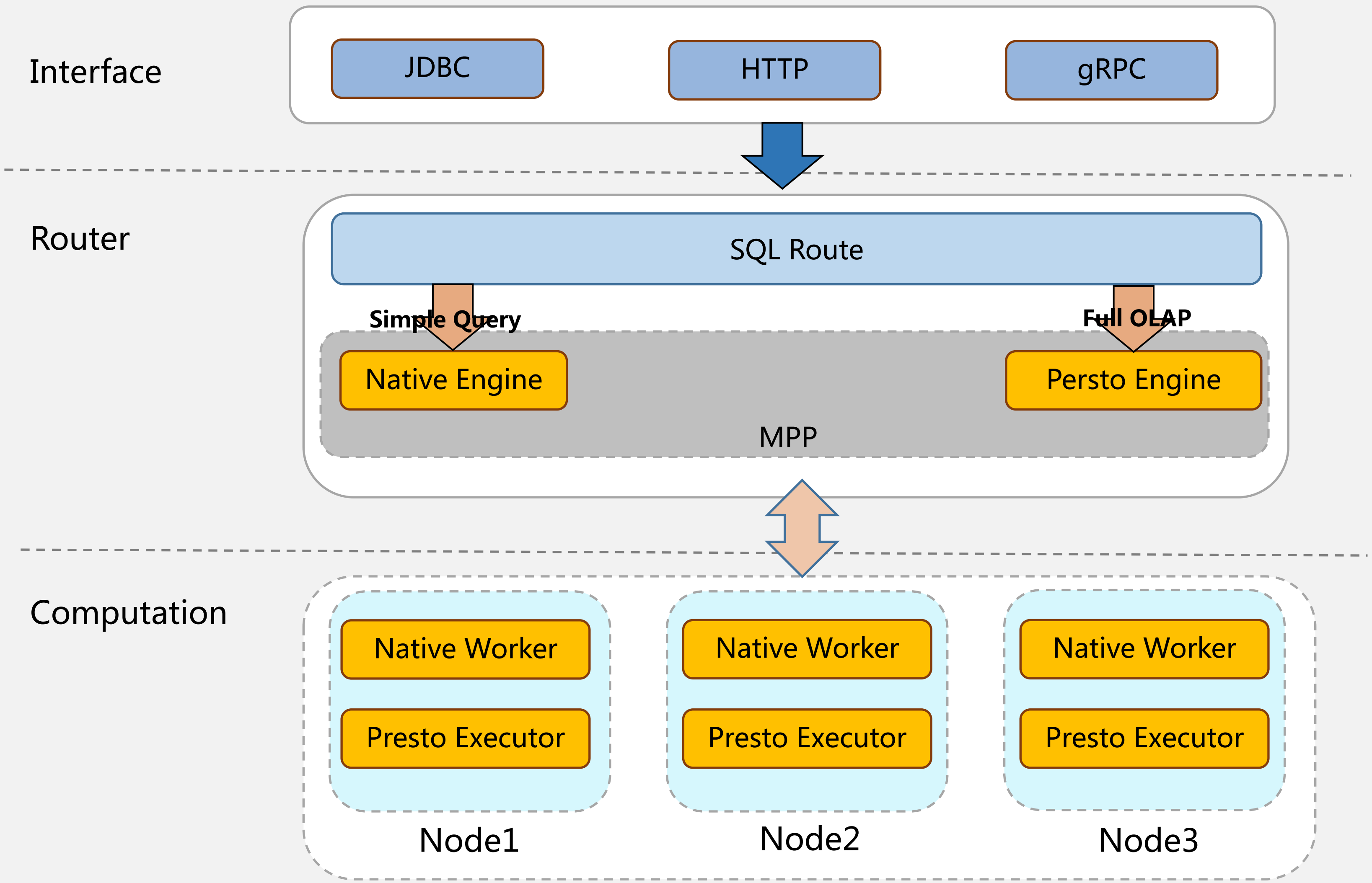
	完全基于 FST 的索引的 key 值索引	基于平衡二叉树的 key 值索引	Lucene 倒排索引
十万次数据点查耗时	490 ms	243 ms	736 ms
相对于 Lucene 倒排索引的存储效果	1.3 倍	1.013 倍	基准 1
千次随机范围查询耗时	142706 ms	13737 ms	24163 ms

LBS(KDB) Index

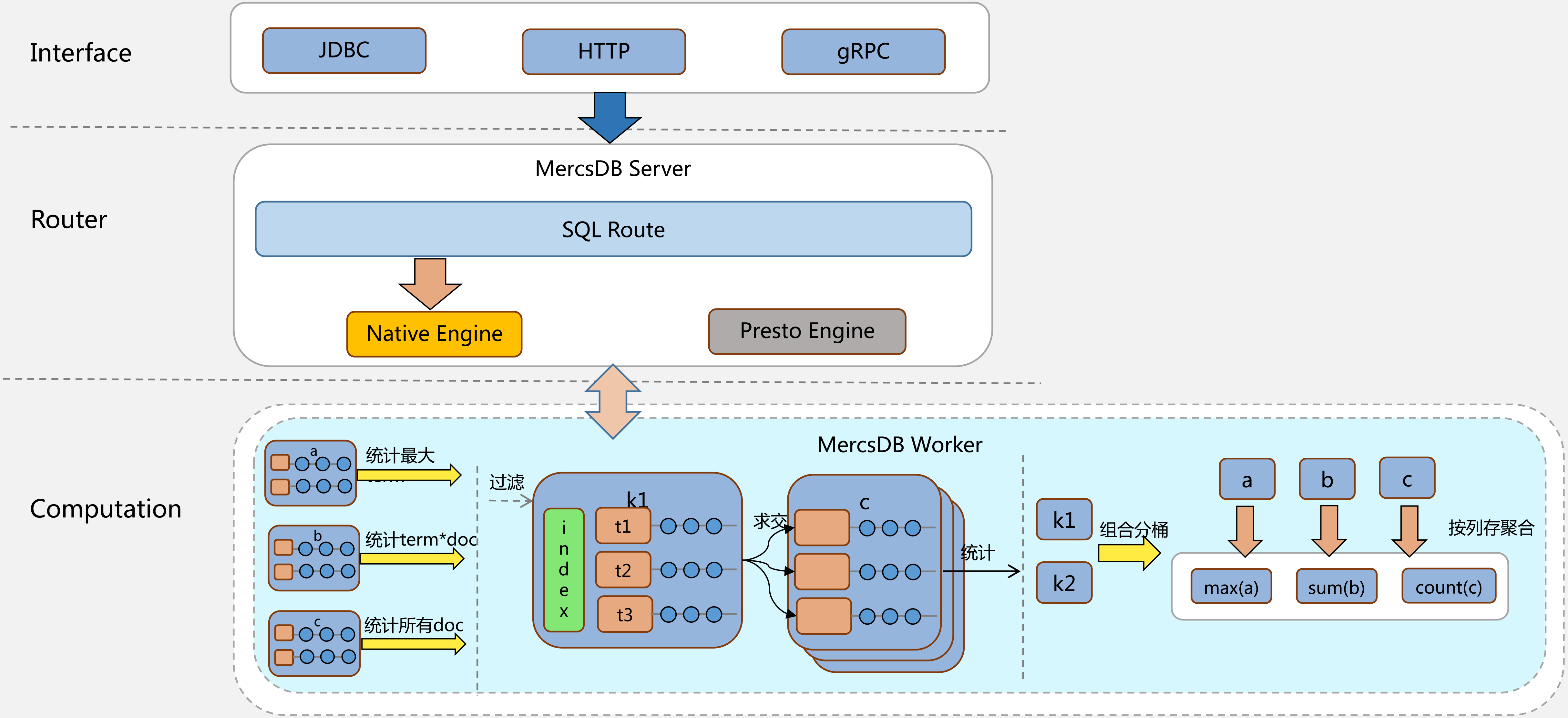


CONTENTS

MercsDB —— Dynamic Router

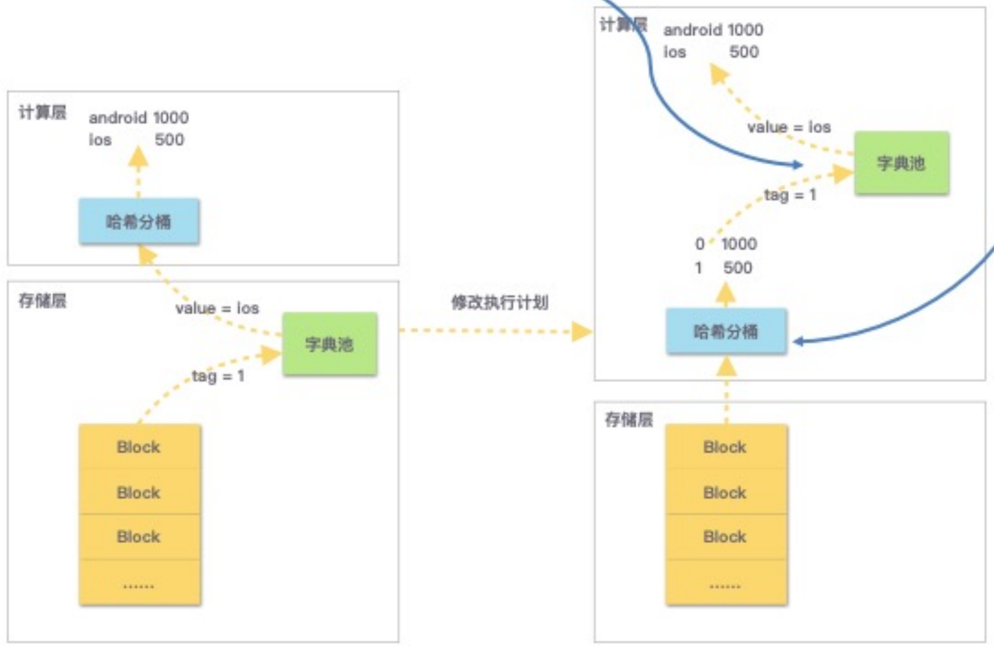


MercsDB Native Engine



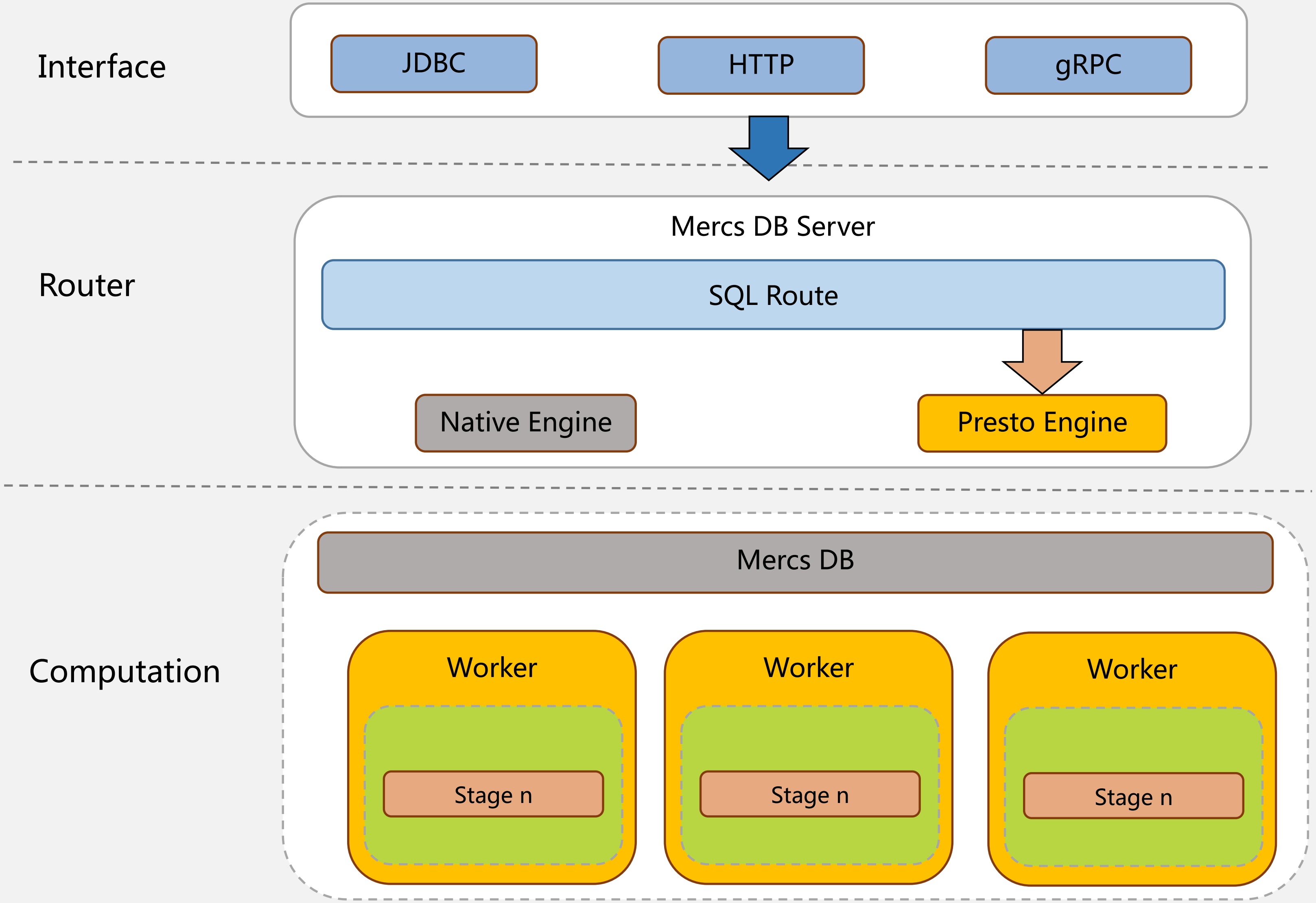
Late Materialization

```
SELECT os ,count(*) FROM table GROUP BY os

SELECT hermesTag(os_tag), count(*) FROM table GROUP BY hermesTag(os_tag)

SELECT hermesTag(os_tag), c FROM (SELECT os_tag, count(*) as c FROM table GROUP BY os_tag)
```



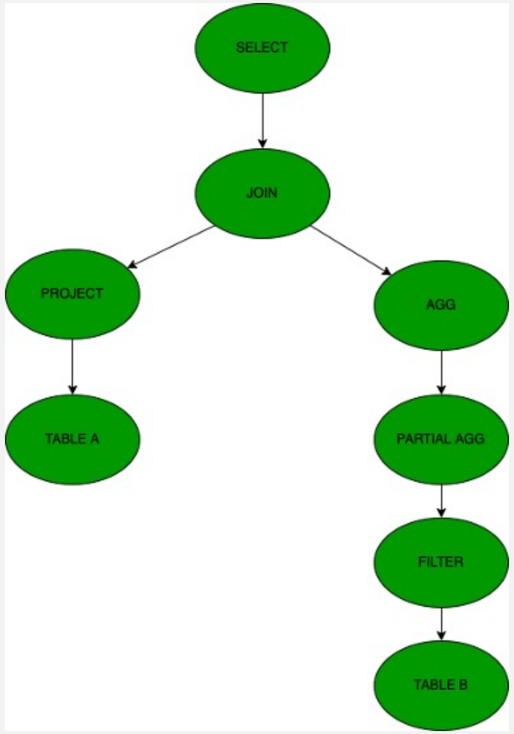
便于理解按照SQL来表达转化，
实际是执行计划中算子的替换
大大提高了分桶计算的效率

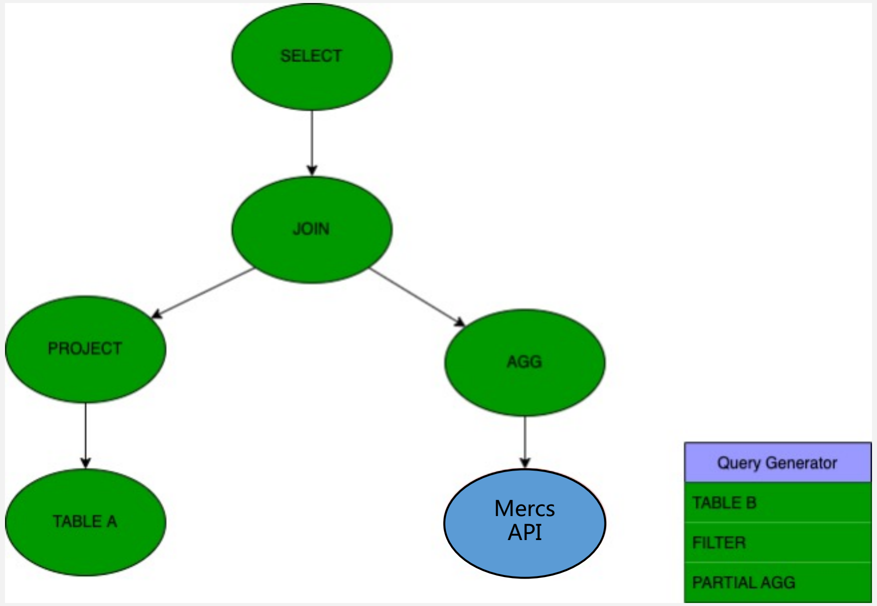
MercsDB Presto Engine



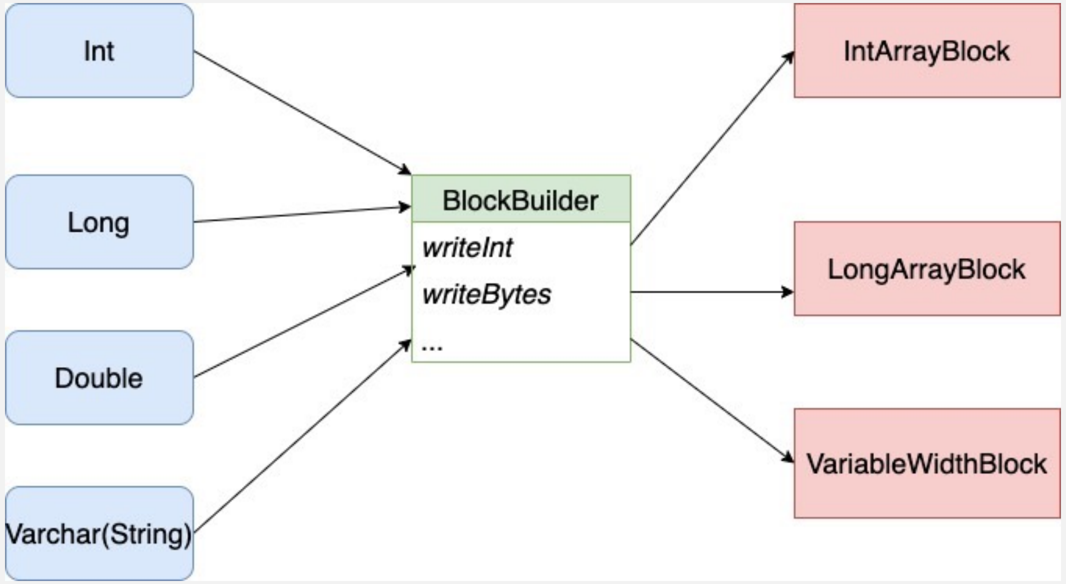
Push down into MercsDB

SELECT * FROM
    (SELECT FROM A)
JOIN
    (SELECT agg FROM B WHERE X)
ON C = D

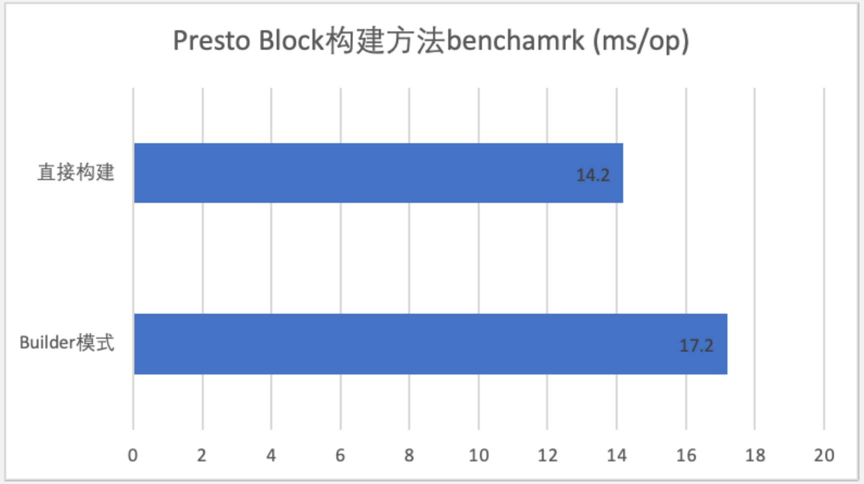




Data transformation



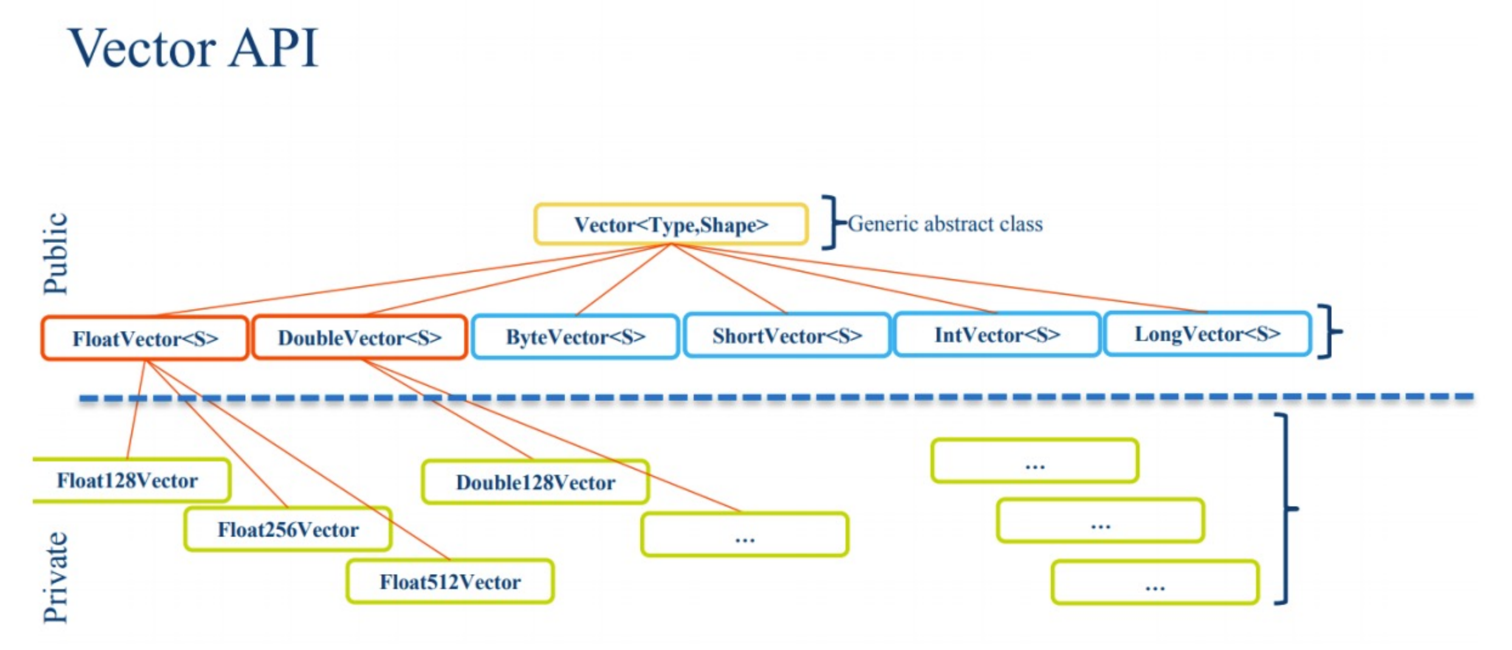
Presto Block	MercsDB Data
long[] Values	Result of Mercs API
bool[] isNulls	Vectorized



Java Vector API

- Incubator since JDK 16
- Tencent KONA JDK 17

```
void scalarComputation(float[] a, float[] b, float[] c) {
   for (int i = 0; i < a.length; i++) {
        c[i] = (a[i] * a[i] + b[i] * b[i]) * -1.0f;
   }
}
```

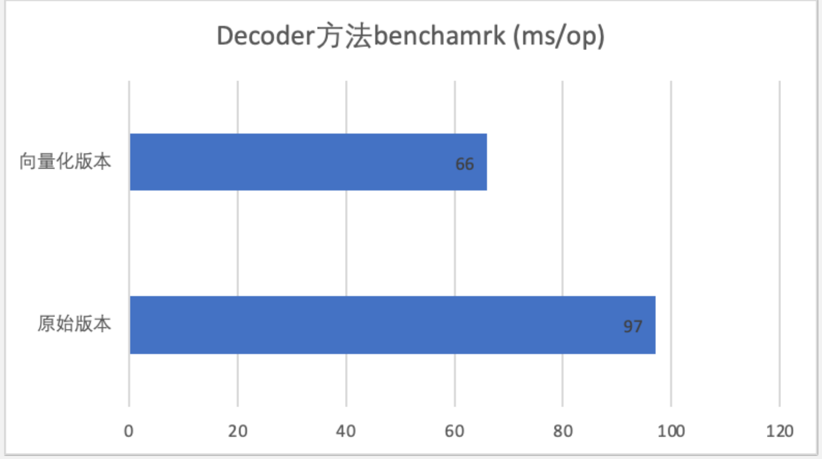



```
static final VectorSpecies<Float> SPECIES = FloatVector.SPECIES_PREFERRED;

void vectorComputation(float[] a, float[] b, float[] c) {
    int i = 0;
    int upperBound = SPECIES.loopBound(a.length);
    for (; i < upperBound; i += SPECIES.length()) {
        // FloatVector va, vb, vc;
        var va = FloatVector.fromArray(SPECIES, a, i);
        var vb = FloatVector.fromArray(SPECIES, b, i);
        var vc = va.mul(va)
                   .add(vb.mul(vb))
                   .neg();
        vc.intoArray(c, i);
    }
    for (; i < a.length; i++) {
        c[i] = (a[i] * a[i] + b[i] * b[i]) * -1.0f;
    }
}
```

Vectorization with Vector API

- Opt1: Unroll Loop
- Opt2: Unify Vector Species
- Opt3:
  - No Boxing & Unboxing
  - No Object creation
  - No function call

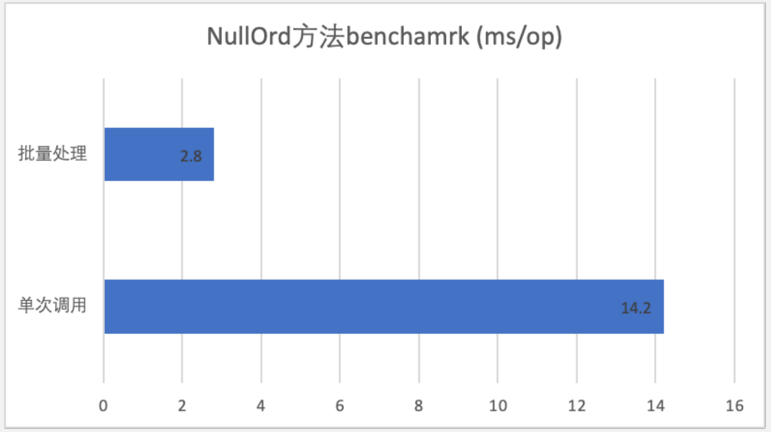
```java
void decode(long[] input, long[] output) {
  for (int i = 0; i < input.length; ++ i) {
    long index = input[i];
    long offset = (index * 20) >>> 3;
    int v = readInt(offset) >>> 8;
    int shift = (int) ( ((index + 1) & 1) << 2);
    output[i] = (v >>> shift) & 0xFFFFF;
  }
}
```



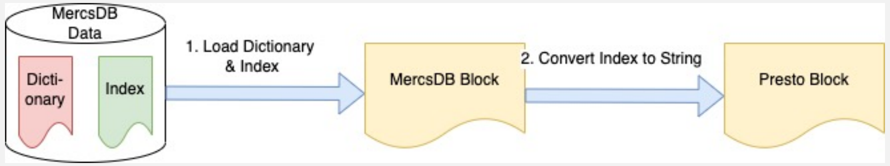

```java
static void decodeVector(long[] input, long[] output) {
    int i;
    int len = input.length - input.length % (4*LONG_SPECIES.length());

    // main loop
    for (i = 0; i < len; i += 4 * LONG_SPECIES.length()) {
        LongVector v1 = LongVector.fromArray(LONG_SPECIES, input, i);
        v1.lanewise(VectorOperators.LSHL,  e: 4).add(v1.lanewise(VectorOperators.LSHL,  e: 2))
                .lanewise(VectorOperators.LSHR,  e: 3).add(currentOffset).intoArray(values,  offset: 0);
        LongVector v2 = LongVector.fromArray(LONG_SPECIES, input,  offset: i + LONG_SPECIES.length());
        v2.lanewise(VectorOperators.LSHL,  e: 4).add(v2.lanewise(VectorOperators.LSHL,  e: 2))
                .lanewise(VectorOperators.LSHR,  e: 3).add(currentOffset).intoArray(values, LONG_SPECIES.length());
        LongVector v3 = LongVector.fromArray(LONG_SPECIES, input,  offset: i + 2 * LONG_SPECIES.length());
        v3.lanewise(VectorOperators.LSHL,  e: 4).add(v3.lanewise(VectorOperators.LSHL,  e: 2))
                .lanewise(VectorOperators.LSHR,  e: 3).add(currentOffset).intoArray(values,  offset: 2 * LONG_SPECIES.length());
        LongVector v4 = LongVector.fromArray(LONG_SPECIES, input,  offset: i + 3 * LONG_SPECIES.length());
        v4.lanewise(VectorOperators.LSHL,  e: 4).add(v4.lanewise(VectorOperators.LSHL,  e: 2))
                .lanewise(VectorOperators.LSHR,  e: 3).add(currentOffset).intoArray(values,  offset: 3 * LONG_SPECIES.length());

        for (int j = 0; j < 4 * LONG_SPECIES.length(); j++) {
            values[j] = readInt(values[j]);
        }

        LongVector.fromArray(LONG_SPECIES, values,  offset: 0)
                .lanewise(VectorOperators.LSHR,  e: 8)
                .lanewise(VectorOperators.LSHR, v1.add(1).and(1).lanewise(VectorOperators.LSHL,  e: 2))
                .and(0xFFFFF).intoArray(output, i);
        LongVector.fromArray(LONG_SPECIES, values, LONG_SPECIES.length())
                .lanewise(VectorOperators.LSHR,  e: 8)
                .lanewise(VectorOperators.LSHR, v2.add(1).and(1).lanewise(VectorOperators.LSHL,  e: 2))
                .and(0xFFFFF).intoArray(output,  offset: i + LONG_SPECIES.length());
        LongVector.fromArray(LONG_SPECIES, values,  offset: 2 * LONG_SPECIES.length())
                .lanewise(VectorOperators.LSHR,  e: 8)
                .lanewise(VectorOperators.LSHR, v3.add(1).and(1).lanewise(VectorOperators.LSHL,  e: 2))
                .and(0xFFFFF).intoArray(output,  offset: i + 2 * LONG_SPECIES.length());
        LongVector.fromArray(LONG_SPECIES, values,  offset: 3 * LONG_SPECIES.length())
                .lanewise(VectorOperators.LSHR,  e: 8)
                .lanewise(VectorOperators.LSHR, v4.add(1).and(1).lanewise(VectorOperators.LSHL,  e: 2))
                .and(0xFFFFF).intoArray(output,  offset: i + 3 * LONG_SPECIES.length());
    }
    // ...
```

Others

- Batch Vectorization



- Sequential memory access



CONTENTS

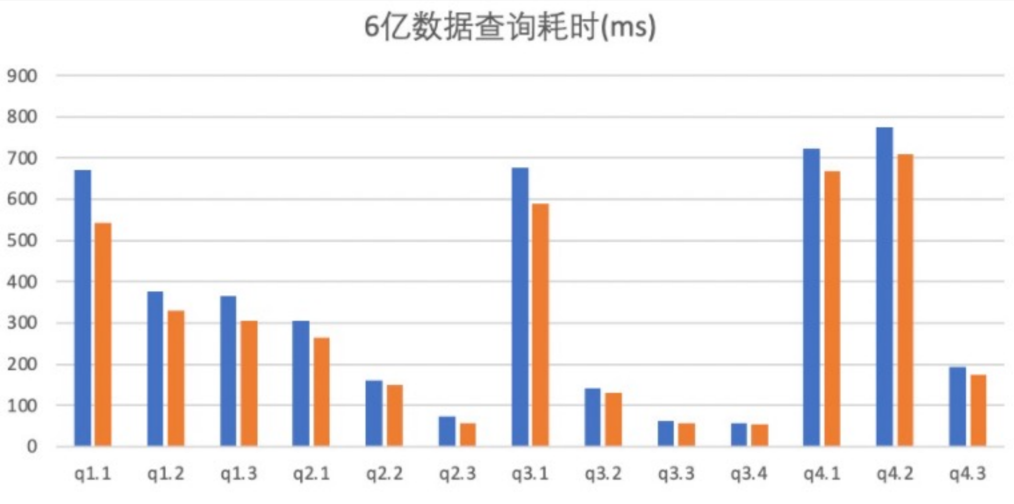
数据平台部

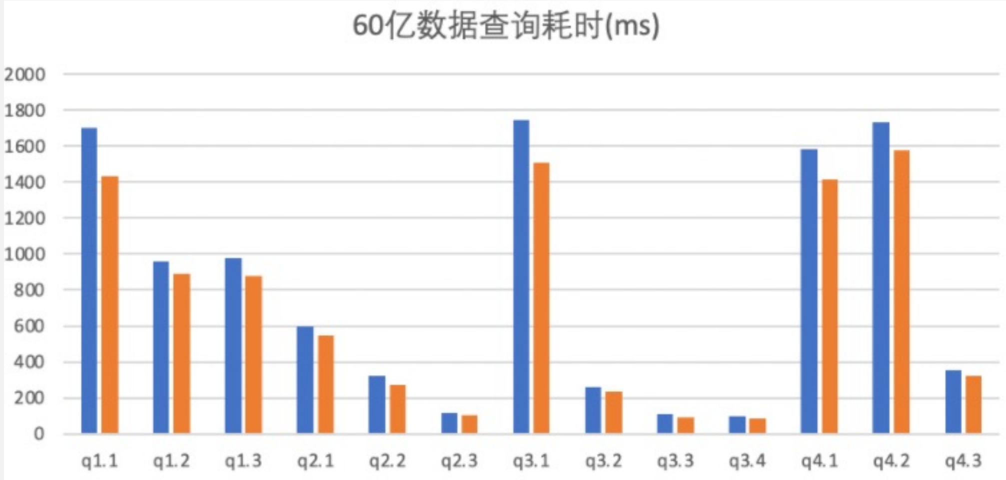
SSB Benchmark 1

Rows	Origin Size	MercsDB (with index)	ClickHouse (with spare and primary-key index)
600M	200GB	89GB	60GB
6B	2TB	882GB	593GB

600M rows

60B rows



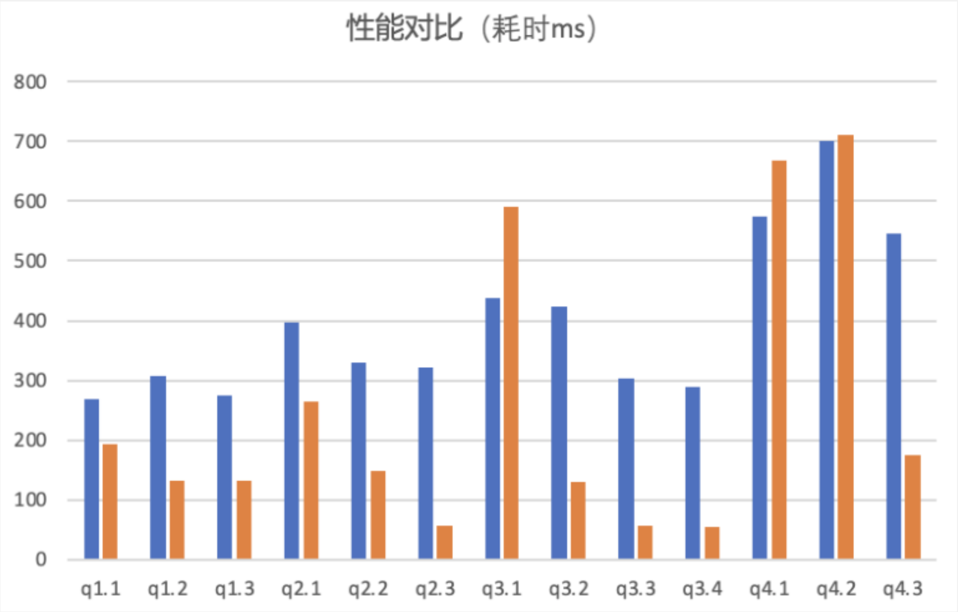


MercsDB

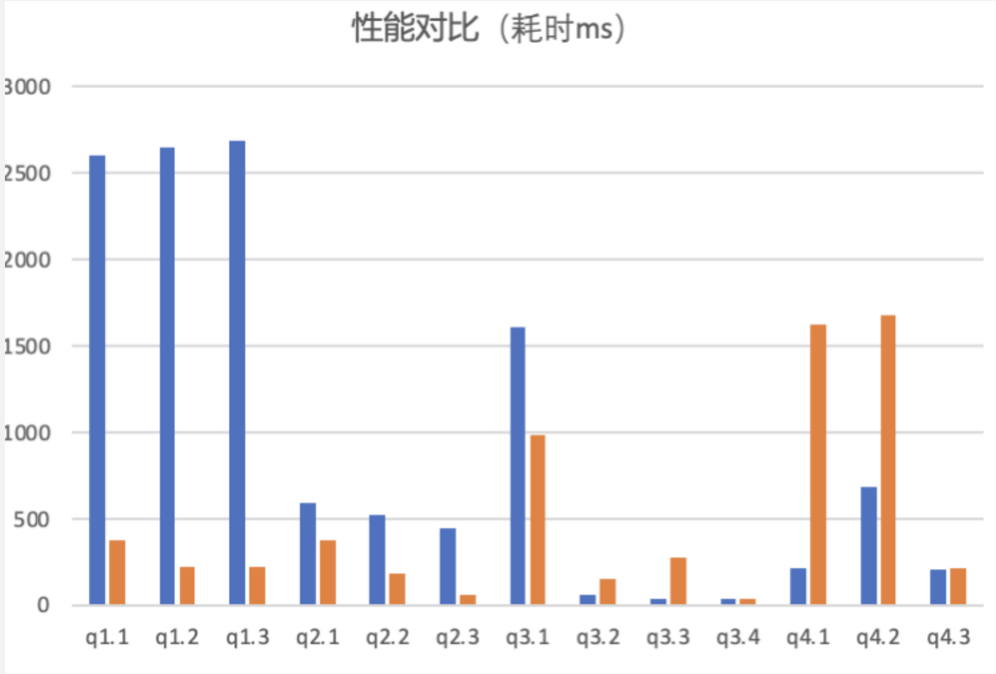
Optimized MercsDB

SSB Benchmark 2: QPS 1

600M rows



60B rows

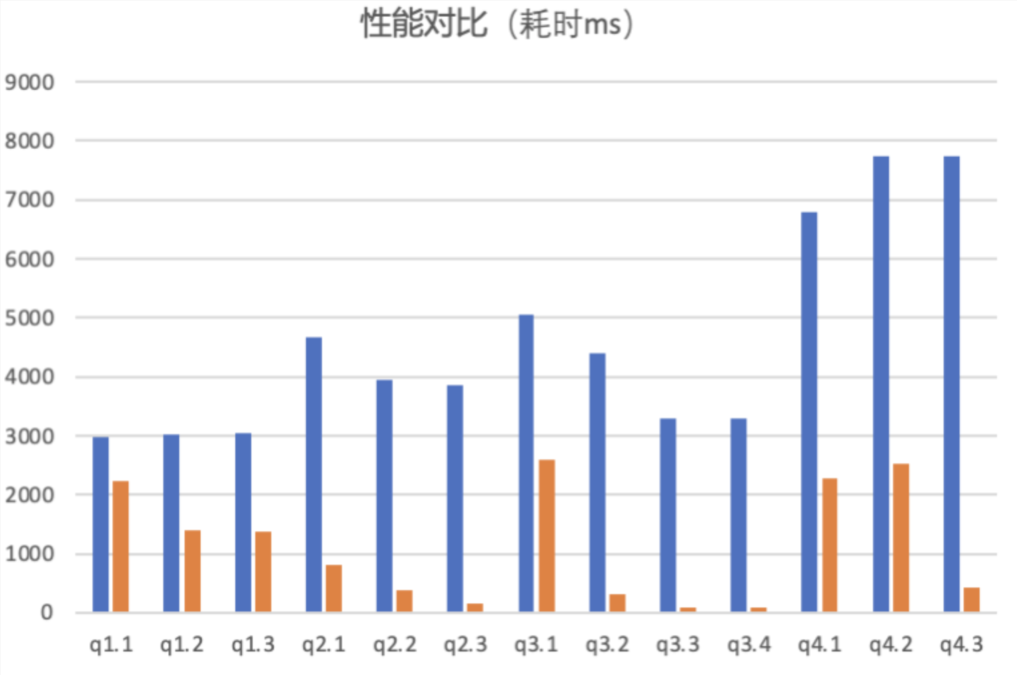


CK

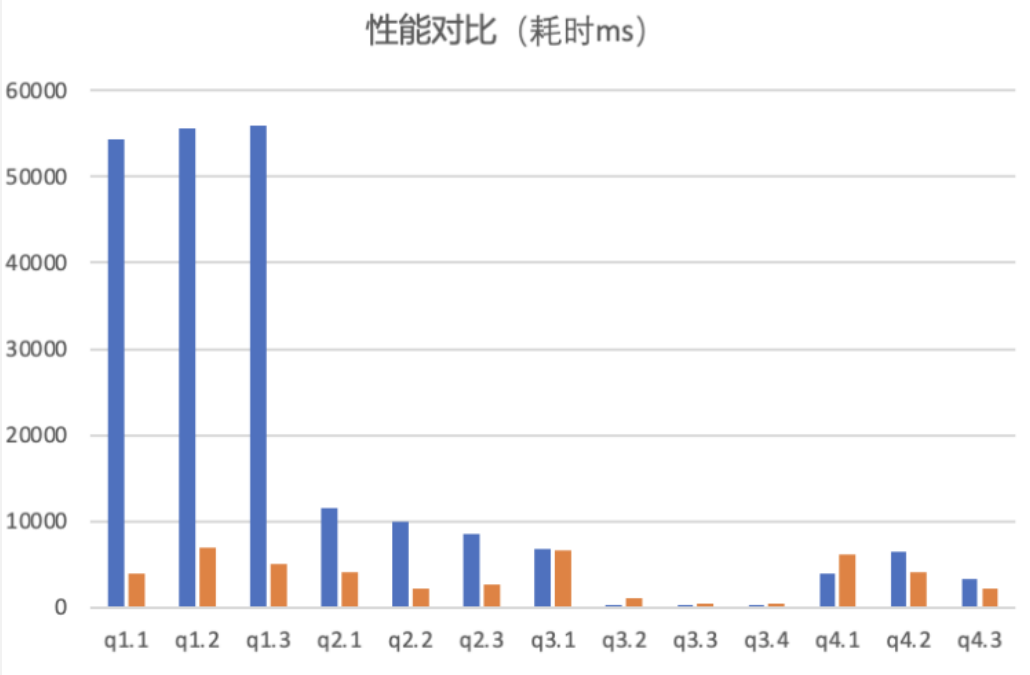
MercsDB

SSB Benchmark 3: QPS 20

600M rows



60B rows



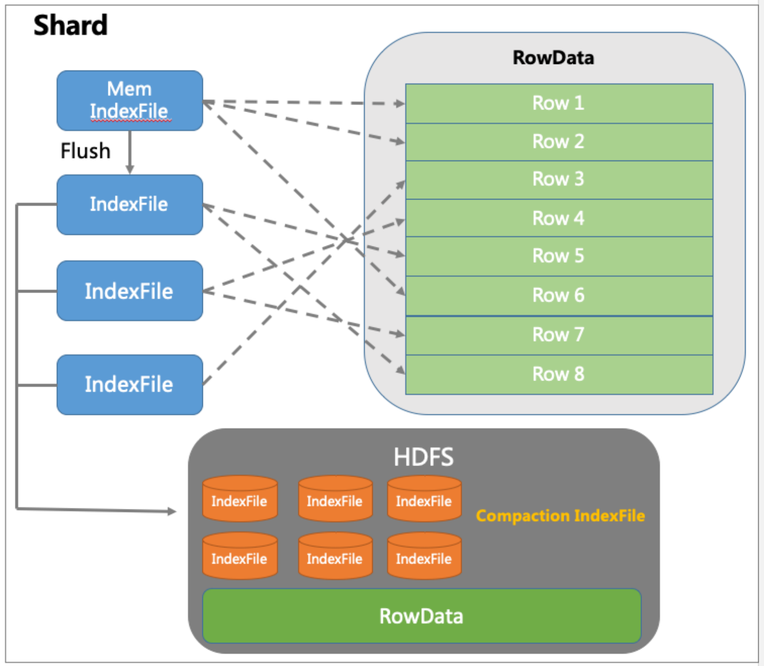
CK

MercsDB

Log Retrieval in WeChat Pay

- Background:
  1. Real Time Write: 100 Billion Rows / day
  2. Retrieval both Global and Specific
  3. Mass Data

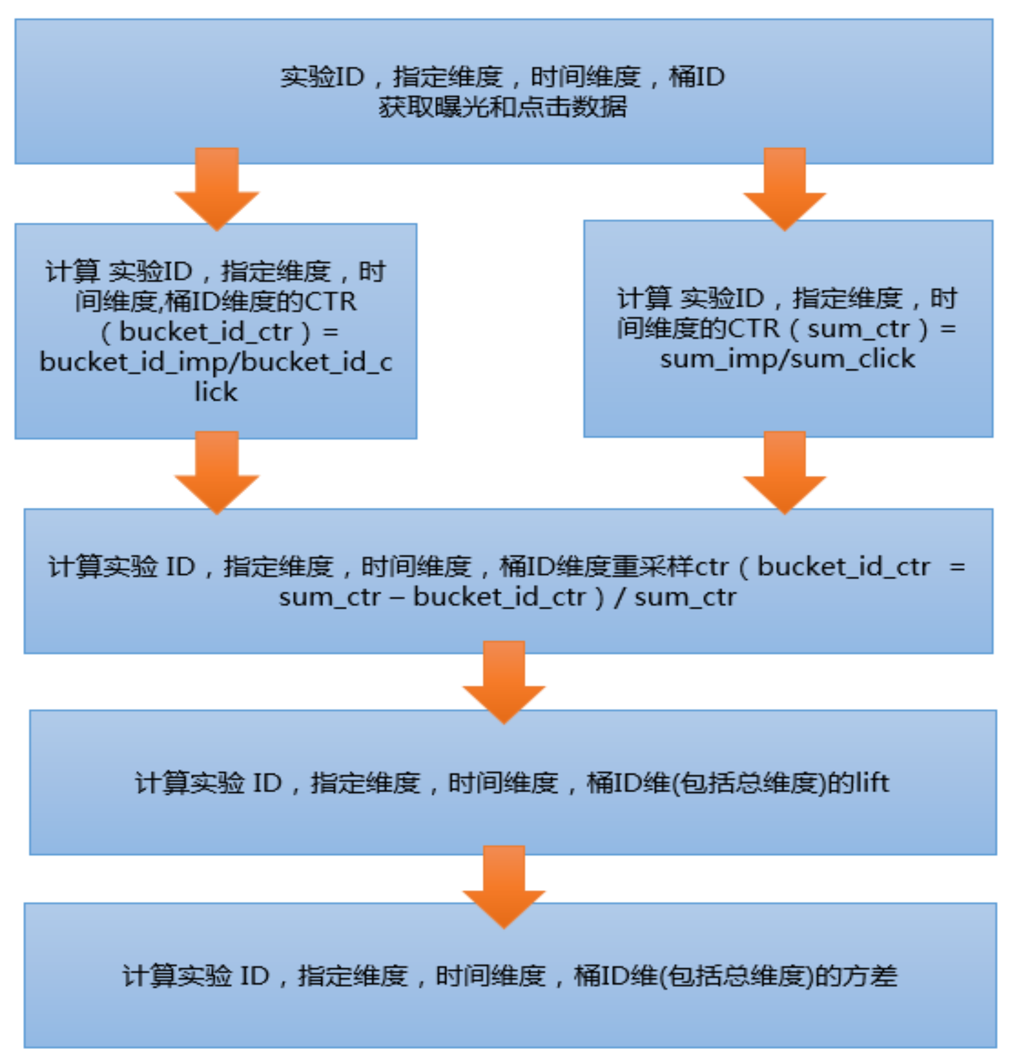
- Solutions:
  1. Write with TubeMQ
  2. Participle and Index on Data
  3. Separation of storage:
     I. IndexFile: Local Disk
     II. RowData: HDFS



AB Test for Ads

- Background:
  1. Second response for Mass Data
  2. Thousand columns in single query
  3. Arbitrary Join

- Solutions:
  1. Sorted by Primary Key (ad_id)
  2. Compressions on metric column, 40% storage of origin data
  3. Presto/Spark supported with Cache



Future Plan

- Cloud Service & Open-Source
- Vectorization
- Fault-Tolerance
- Memory Management
- …

腾讯MercsDB技术探讨

该二维码7天内(6月17日前)有效，重新进入将更新

腾讯大数据

Thanks

数据平台部