实时人群画像
之
高性能OLAP调研

——总悟 2017.12.28

目录

1 Why

用户标签数据＋行为数据，将产生亿级数据

广告业务对用户标签和行为的实时、多维度组合、查询

手动推送、自动推送业务对标签和行为的多维度组合、简单计算查询

~~case by case 接业务~~

我们需要：基于标签的实时人群画像计算引擎

实时性　　多维度组合　　表达式　　服务化

能不能做到亿级标签数据的实时多维度组合查询？查询性能是亚秒级？

业界有没有亚秒级响应的OLAP？如何做到的？？？

我们关注：

面向OLAP场景优化

支持TB级数据

添加新标签成本低

多个标签的任意组合查询，特别是 and /or / not

标签的表达式计算，比如 /　%　+　-　==

查询速度必须快！亚秒级

我们不关注：

事务

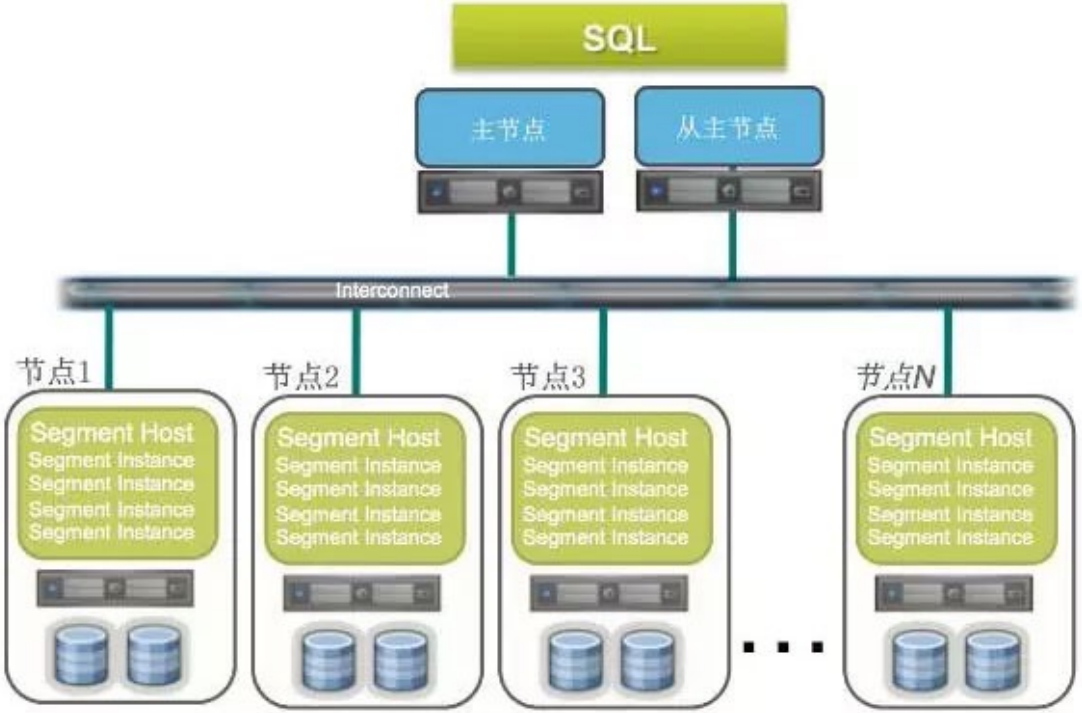
频繁地新增、删除修改

高速写入

top、 limit

2 常见OLAP概览

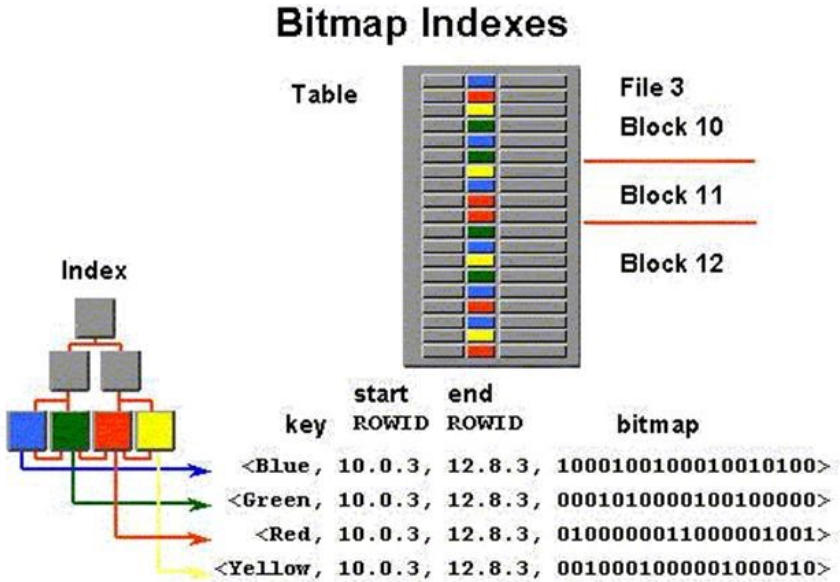
Greenplum 开源MPP（Massively Parallel Processing ）数据库（基于PostgreSQL）



Greenplum

支持列式存储

目前支持B-Tree, GiST, Bitmap三种索引，默认使用B-Tree



Vertica 属于RDBMS，同样采用了MPP架构，主要面向OLAP。

属于商业软件，免费版只允许3个服务节点，1T数据量。特性：

基于C-store设计。

使用列式存储。

多种压缩算法： Run length Encoding、Delta value Encoding、Integer packing for integer data、Block-based dictionary encoding for Character data、Lempel-Ziv compression等。依据数据类型、基数、排序自动选择合适压缩算法进行压缩。

物理存储一个表的多views，称为Projections ,也支持来自多个表的列 。查询时选择合适的Projections。

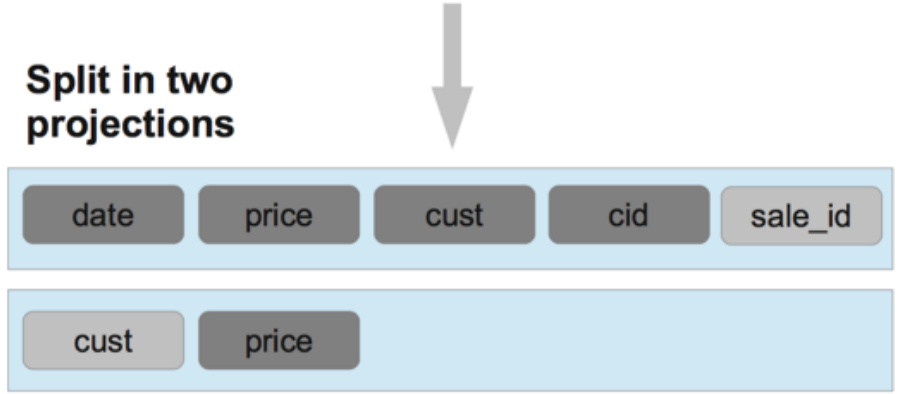
支持标准查询SQL。

集群无master节点，Vertica的元数据存储在每一个节点上。

table and projections

Original Data

sale_id	cid	cust	date	price
1	11	Andrew	01/01/06	$100
2	17	Chuck	01/05/06	$98
3	27	Nga	01/02/06	$90
4	28	Matt	01/03/06	$101
5	89	Ben	01/01/06	$103
1000	89	Ben	01/02/06	$103
1001	11	Andrew	01/03/06	$95



Segmented on several nodes

date	price	cid	cust	sale_id
01/02/06	$90.00	27	Nga	3
01/03/06	$95.00	11	Andrew	1001
01/03/06	$101.00	28	Matt	4

cust	price
Andrew	$95.00
Andrew	$100.00
Chuck	$98.00
Nga	$90.00

Node 1

date	price	cid	cust	sale_id
01/01/06	$100.00	11	Andrew	1
01/01/06	$103.00	89	Ben	5
01/02/06	$103.00	89	Ben	1000
01/05/06	$98.00	17	Chuck	2

cust	price
Ben	$103.00
Ben	$103.00
Matt	$101.00

Node 2

ClickHouse 开源分布式面向分析的高性能数据库

集群模式采用MPP 架构。

读取优化：

- 列式存储，仅处理需要的列。延迟物化
- 索引优化读取位置
- 数据压缩

处理优化：

- Vectorized execution (block-based processing)
- CPU的SIMD提高系统并行处理能力。
- 利用LLVM 做code generation 运行时生成扁平函数代码，减少方法调用的开销。
- 底层代码优化

C++实现，支持聚合查询，支持SQL语法，不支持事务、update、delete。

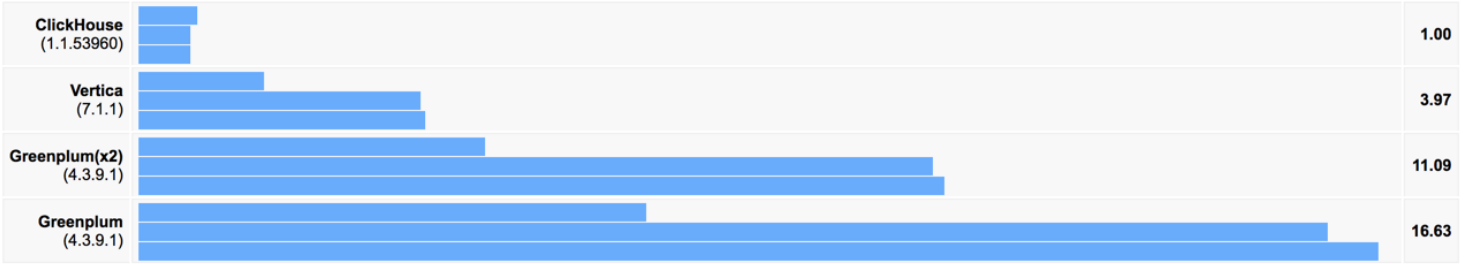
ClickHouse 官网提供的性能对比测试

Compare: ClickHouse | Vertica | Vertica (x3) | Vertica (x6) | InfiniDB | MonetDB | Infobright | Hive | MySQL | MemSQL | Greenplum(x2) | Greenplum

Dataset size: 10 mln. | 100 mln. | 1 bn.

Run number: first (cold cache) | second | third

Relative query processing time (lower is better):



Full results:

Query	ClickHouse (1.1.53960)			Vertica (7.1.1)			Greenplum(x2) (4	
SELECT count() FROM hits	x6.61 (0.297 s.)	(0.011 s.)	(0.011 s.)	(0.045 s.)	x3.04 (0.033 s.)	x2.69 (0.030 s.)	x46.98 (2.110 s.)	x146.36 (1.610
SELECT count() FROM hits WHERE AdvEngine	x1.39 (0.173 s.)	(0.009 s.)	(0.008 s.)	(0.125 s.)	x4.45 (0.044 s.)	x4.81 (0.048 s.)	x10.42 (1.300 s.)	x130.00 (1.300
SELECT sum(AdvEngineID), count(), avg(Resol	x1.08 (0.274 s.)	(0.065 s.)	(0.064 s.)	(0.254 s.)	x2.58 (0.167 s.)	x2.61 (0.167 s.)	x8.64 (2.190 s.)	x32.62 (2.120
SELECT sum(UserID) FROM hits	x2.07 (0.552 s.)	(0.046 s.)	(0.045 s.)	(0.267 s.)	x1.32 (0.061 s.)	x1.32 (0.059 s.)	x9.29 (2.480 s.)	x55.00 (2.530
SELECT uniq(UserID) FROM hits	(0.485 s.)	(0.097 s.)	(0.096 s.)	x1.86 (0.901 s.)	x9.08 (0.881 s.)	x10.75 (1.032 s.)	x6.10 (2.960 s.)	x31.44 (3.050
SELECT uniq(SearchPhrase) FROM hits	(0.625 s.)	(0.211 s.)	(0.204 s.)	x2.05 (1.284 s.)	x4.70 (0.991 s.)	x4.96 (1.012 s.)	x6.43 (4.020 s.)	x19.05 (4.020
SELECT min(EventDate), max(EventDate) FROI	x1.09 (0.164 s.)	(0.047 s.)	(0.044 s.)	(0.150 s.)	x1.45 (0.068 s.)	x1.53 (0.067 s.)	x13.28 (1.990 s.)	x38.09 (1.790
SELECT AdvEngineID, count() FROM hits WHE	(0.056 s.)	(0.009 s.)	(0.009 s.)	x2.66 (0.149 s.)	x8.31 (0.083 s.)	x7.82 (0.078 s.)	x76.07 (4.260 s.)	x377.00 (3.770

ClickHouse表引擎MergeTree

MergeTree(EventDate,(CounterID,EventDate),8192)

分区列　　主键（可多个）　　index_granularity

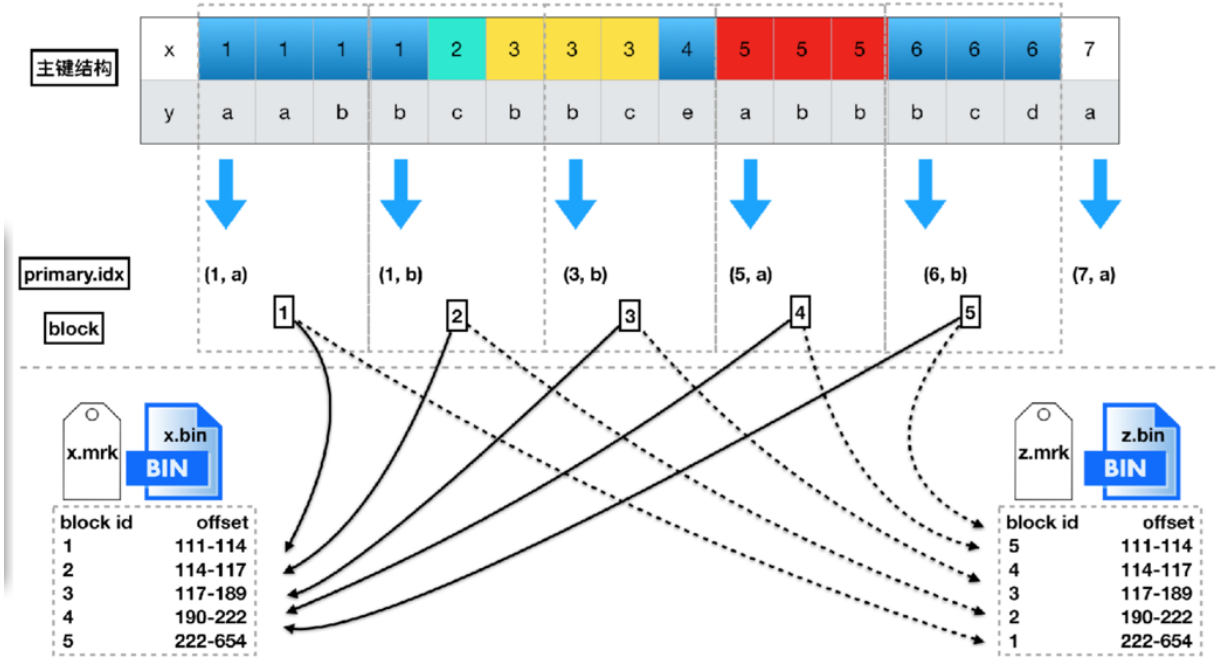
写优化：

创建block 写入数据，异步merge 多个block，按照主键排序，不支持删除修改。

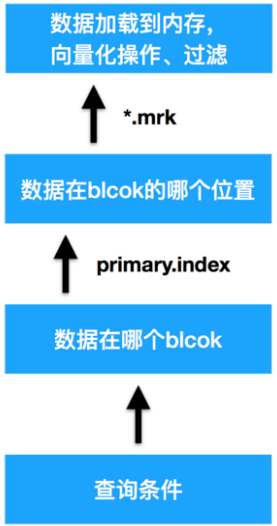
block 大小等于index_granularity（稀疏索引粒度）

思路有点类似于 LSM Tree，但是直接批量写入磁盘，没有记log，没有内存表

索引


主键结构
x
1 1 1 1 2 3 3 3 4 5 5 5 6 6 6 7
y
a a b b c b b c e a b b b c d a
primary.idx
(1, a)
(1, b)
(3, b)
(5, a)
(6, b)
(7, a)
block
1
2
3
4
5
x.mrk
x.bin
BIN
block id offset
1 111-114
2 114-117
3 117-189
4 190-222
5 222-654
z.mrk
z.bin
BIN
block id offset
5 111-114
4 114-117
3 117-189
2 190-222
1 222-654

查询流程



全主键

where x='3' and y='c'

1. 判断，只需扫描block 2、3（定位block）
2. 使用mrk文件，定位到数据（找到数据）
3.加载内存过滤
4. 返回
5. y的作用呢？如果是(1, c) (1, c) (1, d) (1, e)呢？

半主键

where x='3'
只扫描block 2、3

where y='c'
1. block 1首先被过滤掉（(1,a) (1,b) 1=1）
2. 所需block 2、3、4、5 （定位block）
3. 剩余过程类似
4. 该情况下，存在过滤效果差的情况

非主键

where z='?'
如何定位z的数据?
等效于
where x=any and y=any and z='?'
1. 所有block（定位block）
2. 取所有mrk里所有的数据偏移指向，
即全扫描
3. 过滤

主键+非主键

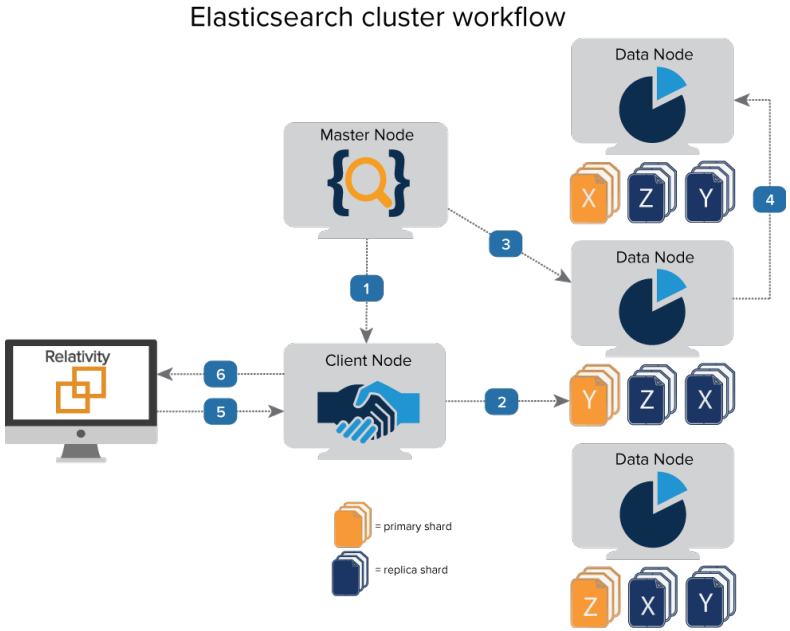
where x='?' and z='?'

1. 利用主键x，找到x的block，同时也一定是z要过滤的block（定位block）
2. 取出x、z mrk文件里的偏移量（定位数据）
3. 加载、过滤
4. 返回

利用SIMD扫描

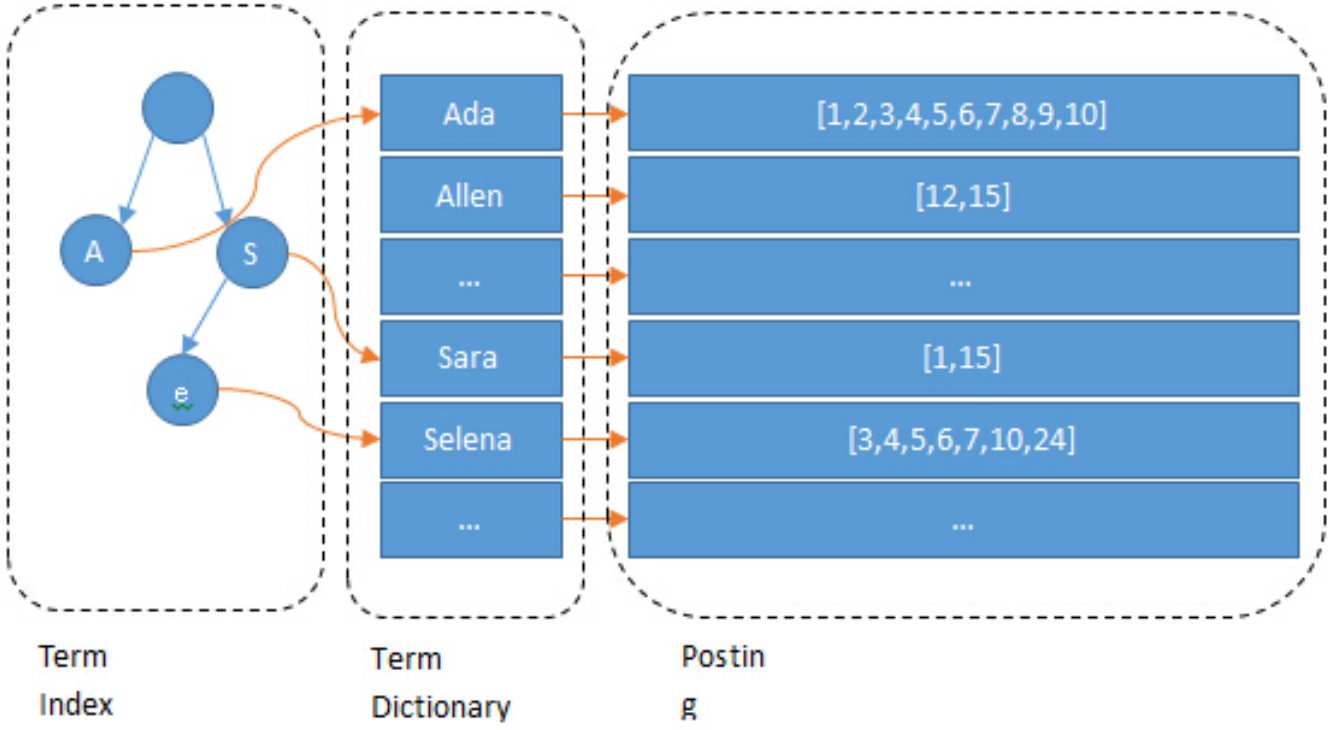
过多的主键，对查询性能并没有太大的影响。

Elastic Search 分布式检索与分析引擎。搜索存储功能基于 Lucene，ES在其之上封装了索引查询和分布式相关功能。

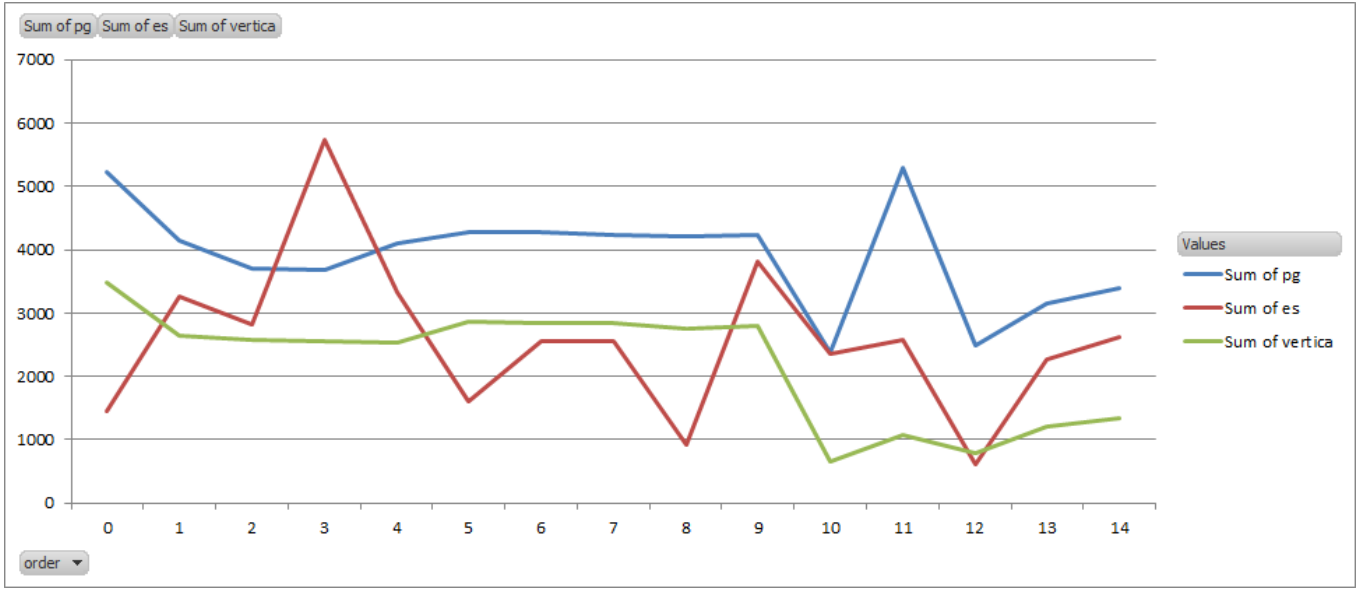


Elastic Search 开源全文检索与分析引擎

索引原理



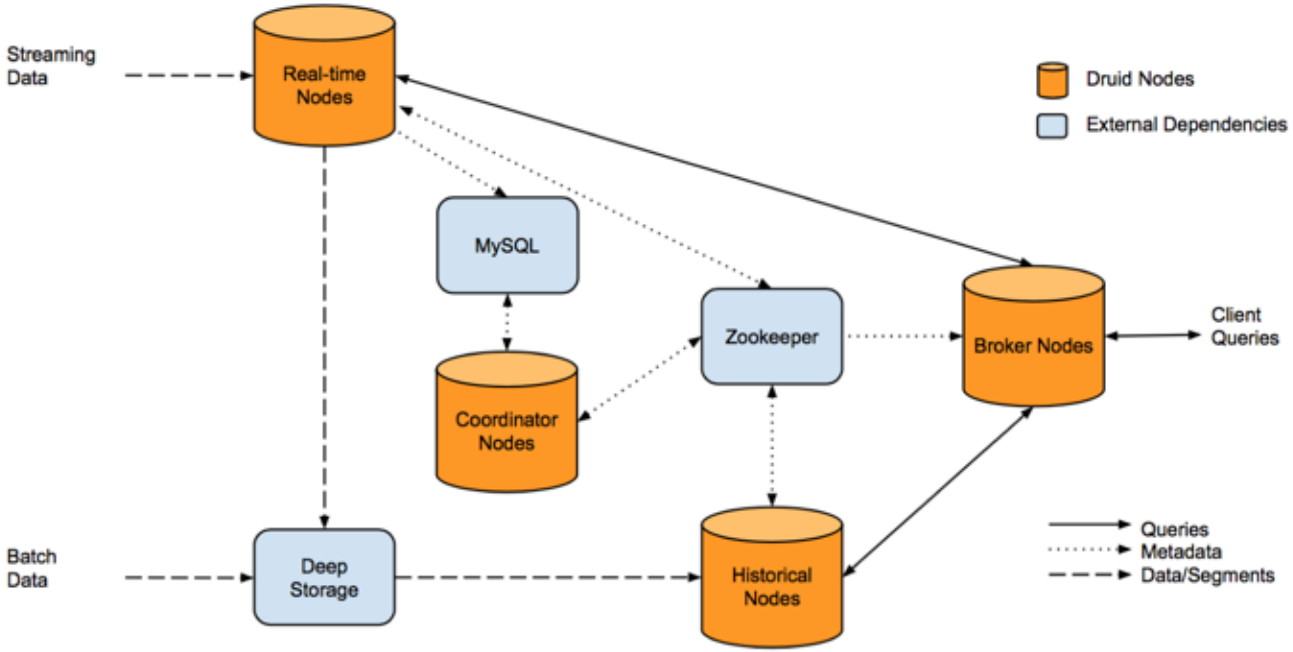
使用业务数据做的性能测试



1 亿条用户标签数据。

4 核 内存:8 GB 磁盘 500G SSD * 3

Druid 是一个开源的、面向海量数据实时查询与分析的OLAP数据存储
采用lambda架构。非常适合处理事件类数据。



Druid 数据分片叫segment，内部数据以时间分割，数据以列式存储，为聚合的列做索引。

Segment含有三种类型的数据结构

Timestamp　　Dimensions　　Metrics

Timestamp	Page	Username	Gender	City	Characters Added	Characters Removed
2011-01-01T01:00:00Z	Justin Bieber	Boxer	Male	San Francisco	1800	25
2011-01-01T01:00:00Z	Justin Bieber	Reach	Male	Waterloo	2912	42
2011-01-01T02:00:00Z	Ke$ha	Helz	Male	Calgary	1953	17
2011-01-01T02:00:00Z	Ke$ha	Xeno	Male	Taiyuan	3194	170

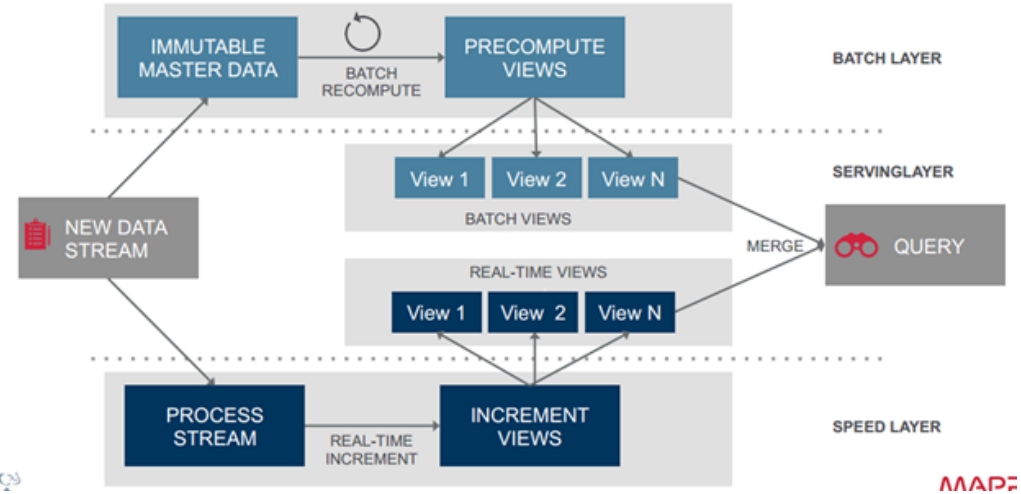
Timestamp和Metric列使用LZ4压缩

每个Dimensions列使用dictionary编码＋bitmap index 倒排索引加速查询。

Lambda架构介绍

Nathan Marz 定义： 数据系统 = 数据 + 查询
Query = function(all data)
数据量太大，如果要支持实时查询，需要预计算

batch view = function(all data)
realtime view = function(realtime view, new data)
query = function(batch view .realtime view)



Pilosa 开源分布式位图索引
全内存计算。使用bitmap matrix索引数据，底层使用RoaringBitmap 数据结构，压缩比较高。
提供了BSI (Bit-Sliced Indexing) 内部实现了将高基数int类型n-bit的数据转换成n＋1个bitmaps表示。 方便构建高基数的 bitmap index，适合多维度查询。

持久化bitmap文件（roaring bitmap格式的文件）到磁盘，通过mmap加载到内存。
go语言实现，有go、java 、python client 。
可扩展元数据管理，部分计算可转化到元数据计算上。

支持集群部署，自定义 JSON 查询语句。
同等规模的数据量下查询性能比Elastic Search 快5～10倍。

Bitmap

对于 1，3，7，15 这4个int值使用bitmap存储

10100010 00000010

只需要2个byte

1亿个int型数字，正常存储需要380M，使用bitmap存储只需要12.5M。

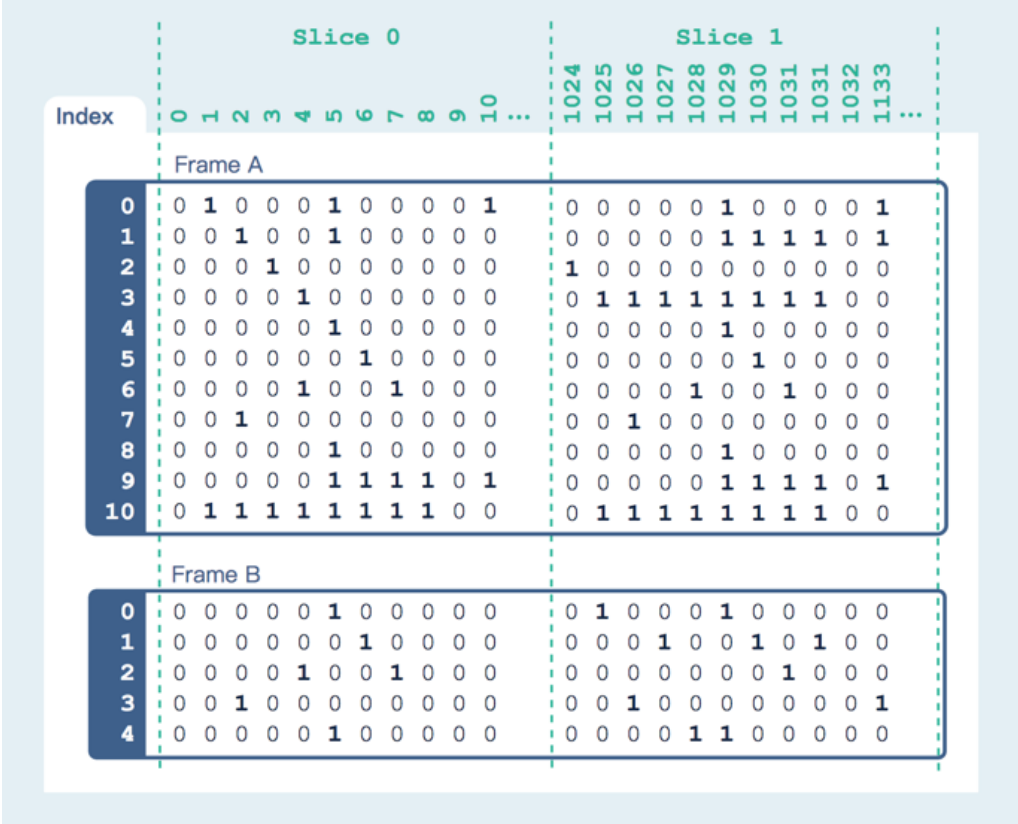
对Bitmap编码压缩： Run Length Encoding（游程编码）、RoaringBitmap 而且不需要解压直接支持bitset运算。

由Bitmap表示的数据可以全部加载到内存中。直接支持 bitset运算 : and /or/xor/not

对数据的多维度查询，转换为2个维度的bitmap数组 做bitset运算，计算速度非常快。

字符型数据可以通过字典编码为数字类型。

Pilosa 数据模型


Slice 0
Slice 1
Index
0 1 2 3 4 5 6 7 8 9 10 ...
1024 1025 1026 1027 1028 1029 1030 1031 1031 1032 1133 ...
Frame A
0 0 1 0 0 0 1 0 0 0 0 1 | 0 0 0 0 0 1 0 0 0 0 1
1 0 0 1 0 0 1 0 0 0 0 0 | 0 0 0 0 0 1 1 1 1 0 1
2 0 0 0 1 0 0 0 0 0 0 0 | 1 0 0 0 0 0 0 0 0 0 0
3 0 0 0 0 1 0 0 0 0 0 0 | 0 1 1 1 1 1 1 1 1 0 0
4 0 0 0 0 0 1 0 0 0 0 0 | 0 0 0 0 0 1 0 0 0 0 0
5 0 0 0 0 0 0 1 0 0 0 0 | 0 0 0 0 0 0 1 0 0 0 0
6 0 0 0 0 1 0 0 1 0 0 0 | 0 0 0 0 1 0 0 1 0 0 0
7 0 0 1 0 0 0 0 0 0 0 0 | 0 0 1 0 0 0 0 0 0 0 0
8 0 0 0 0 0 1 0 0 0 0 0 | 0 0 0 0 0 1 0 0 0 0 0
9 0 0 0 0 0 1 1 1 1 0 1 | 0 0 0 0 0 1 1 1 1 0 1
10 0 1 1 1 1 1 1 1 1 0 0 | 0 1 1 1 1 1 1 1 1 0 0
Frame B
0 0 0 0 0 0 1 0 0 0 0 0 | 0 1 0 0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 1 0 0 0 0 | 0 0 0 1 0 0 1 0 1 0 0
2 0 0 0 0 1 0 0 1 0 0 0 | 0 0 0 0 0 0 0 1 0 0 0
3 0 0 1 0 0 0 0 0 0 0 0 | 0 0 1 0 0 0 0 0 0 0 1
4 0 0 0 0 0 1 0 0 0 0 0 | 0 0 0 0 1 1 0 0 0 0 0

Pilosa 实战：

8 core 32G memory

2500w用户的19个 tag 数据（4G）导入Pilosa，占用内存900M~1.2G。

每个标签创建一个frame,标签值作为row_id ， user_id 做为column_id

```
[root@iZ23o7f05j8Z pilosa]# curl  -s -w %{time_total}s"\n" localhost:10101/index/repository/query      -X POST      -d 'C
ount(Bitmap(frame="i_tag_4",  rowID=320000))'
{"results":[1871575]}
0.007s
[root@iZ23o7f05j8Z pilosa]# curl  -s -w %{time_total}s"\n" localhost:10101/index/repository/query  \
>     -X POST \
>     -d 'Count(Intersect( Bitmap(frame="i_tag_1",rowID=0),Bitmap(frame="i_tag_11",rowID=320100),Bitmap(frame="i_tag_33",
rowID=4),Bitmap(frame="i_tag_4",rowID=320000),Bitmap(frame="i_tag_11",rowID=320100),Bitmap(frame="i_tag_176",rowID=201710
13),Bitmap(frame="i_tag_2564",rowID=440000)))'
{"results":[5]}
0.016s
[root@iZ23o7f05j8Z pilosa]# curl  -s -w %{time_total}s"\n" localhost:10101/index/repository/query  \
>   -X POST \
>   -d 'Intersect( Bitmap(frame="i_tag_1",rowID=0),Bitmap(frame="i_tag_11",rowID=320100),Bitmap(frame="i_tag_33",rowID=4)
,Bitmap(frame="i_tag_4",rowID=320000),Bitmap(frame="i_tag_11",rowID=320100),Bitmap(frame="i_tag_176",rowID=20171013),Bitm
ap(frame="i_tag_2564",rowID=440000))'
{"results":[{"attrs":{},"bits":[69408904,91532099,137617644,146696850,169898924]}]}
0.192s
[root@iZ23o7f05j8Z pilosa]#
```

Summary of the 1.1 Billion Taxi Rides Benchmarks

Query 1	Query 2	Query 3	Query 4	Setup
0.005	**1.505**	**0.177**	**12.86**	**Pilosa, 3-node c4.8xlarge**
0.021	0.053	0.165	0.51	MapD & 8 Nvidia Pascal Titan Xs
0.027	0.083	0.163	0.891	MapD & 8 Nvidia Tesla K80s
0.028	0.2	0.237	0.578	MapD & 4-node g2.8xlarge cluster
0.034	0.061	0.178	0.498	MapD & 2-node p2.xlarge cluster
0.036	0.131	0.439	0.964	MapD & 4 Nvidia Titan Xs
0.051	0.146	0.047	0.794	kdb+/q & 4 Intel Xeon Phi 7210 CPUs
1.034	3.058	5.354	12.748	ClickHouse, Intel Core i5 4670K
1.56	1.25	2.25	2.97	Redshift, 6-node ds2.8xlarge cluster
2	2	1	3	BigQuery
4	4	10	21	Presto, 50-node n1-standard-4 cluster
6.41	6.19	6.09	6.63	Amazon Athena
8.1	18.18	n/a	n/a	Elasticsearch (heavily tuned)
10.19	8.134	19.624	85.942	Spark 2.1, 11 x m3.xlarge cluster w/ HDFS
11	10	21	31	Presto, 10-node n1-standard-4 cluster
14.389	32.148	33.448	67.312	Vertica, Intel Core i5 4670K
34.48	63.3	n/a	n/a	Elasticsearch (lightly tuned)
35	39	64	81	Presto, 5-node m3.xlarge cluster w/ HDFS
43	45	27	44	Presto, 50-node m3.xlarge cluster w/ S3
152	175	235	368	PostgreSQL 9.5 & cstore_fdw
264	313	620	961	Spark 1.6, 5-node m3.xlarge cluster w/ S3

Query1:

```
SELECT cab_type,
       count(*)
FROM trips
GROUP BY cab_type;
```

Query2:

```
SELECT passenger_count,
       avg(total_amount)
FROM trips
GROUP BY passenger_count;
```

Query3:

```
SELECT passenger_count,
       year(pickup_datetime),
       count(*)
FROM trips
GROUP BY passenger_count,
         year(pickup_datetime);
```

Query4:

```
SELECT passenger_count,
       year(pickup_datetime) trip_year,
       round(trip_distance),
       count(*) trips
FROM trips
GROUP BY passenger_count,
         year(pickup_datetime),
         round(trip_distance)
ORDER BY trip_year,
         trips desc;
```

小结

从硬件资源角度看：

CPU：SIMD 单指令多数据集;利用LLVM代码生成，减少函数堆栈调用导致的cpu指令开销；向量化执行引擎（Vectored iterator model）提升cpu cache的命中率

内存：全内存计算；减少不必要的数据加载

磁盘：列式存储，数据压缩，减少磁盘IO，mmap加载

其他：GPU并行运算

从软件设计角度看：

MPP架构，不可变数据，存储与计算紧密耦合，舍弃不常用特性，高效索引

Why?

Latency Comparison Numbers for 2017

L1 cache reference	1 ns		
L2 cache reference	4 ns		
Main memory reference	100 ns		
Send 2K bytes over 1 commodity network	125 ns		
randomly from SSD*	16,000 ns	16 us	
Read 1 MB sequentially from SSD	98,000 ns	98 us	
Read 1 MB sequentially from memory	6000 ns	6 us	
Disk seek	3,000,000 ns	3,000 us	3 ms
Read 1 MB sequentially from disk	1,000,000 ns	1,000 us	1 ms

计算并行化

减少IO

3 选择

回顾一下需求特点：多维查询，查询表达式复杂，实时性要求

大方向：MPP架构、列式、基于bitmap index、元数据管理、集群分片

Pilosa VS Druid VS 自己撸?

如果自己撸，实现上最终会类似于Pilosa

技术选型还要考虑：

人员成本

学习成本

开发效率

可维护度

局限性

性能极限

参考文献：

https://weibo.com/ttarticle/p/show?id=2309404084808111288473
https://yq.aliyun.com/articles/80565
http://editorup.zol.com.cn/upload/201402/532c100ccd7d3.pdf
The Vertica Analytic Database C-Store 7 Years Later
MySQL DBA解锁数据分析的新姿势-ClickHouse-新浪-高鹏
https://clickhouse.yandex/docs/en/single/index.html#document-table_engines/mergetree
https://clickhouse.yandex/
Quick Tour of ClickHouse Internals
http://www.infoq.com/cn/articles/database-timestamp-02
http://static.druid.io/docs/druid.pdf
http://druid.io/docs/0.11.0/design/index.html
http://zqhxuyuan.github.io/2015/12/03/2015-12-03-Druid-Design
http://blog.csdn.net/lvsaixia/article/details/51778487
https://www.pilosa.com/docs/latest/introduction/
http://tech.marksblogg.com/benchmarks.html
https://www.pilosa.com/blog/billion-taxi-ride-dataset-with-pilosa

Thank you