OceanBase
A Powerful Database System

OceanBase的SQL优化器和分布式并行执行

关于OceanBase

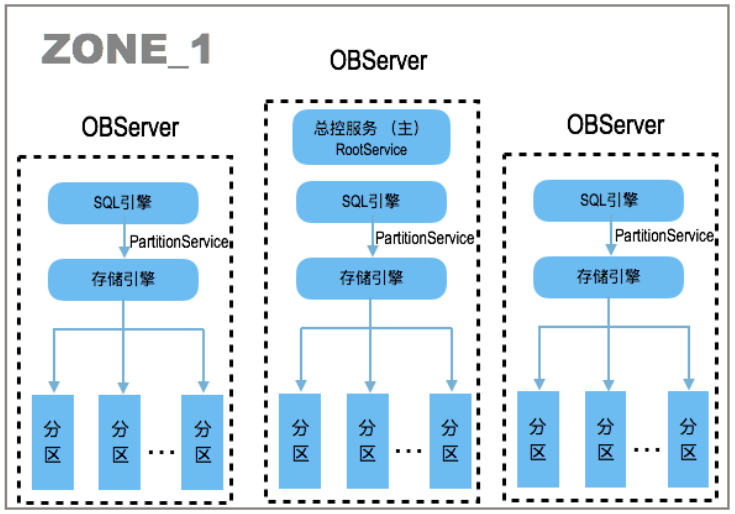
- 阿里巴巴、蚂蚁金服自主研发的通用关系型数据库
- 分布式架构下保证金融级数据一致性
- 高可用、线性扩展、弹性部署、低成本、高性能
- 支持蚂蚁金服100%核心交易系统
- 稳定支撑阿里/蚂蚁内部上百个关键业务以及浙商银行、南京银行等多个外部客户

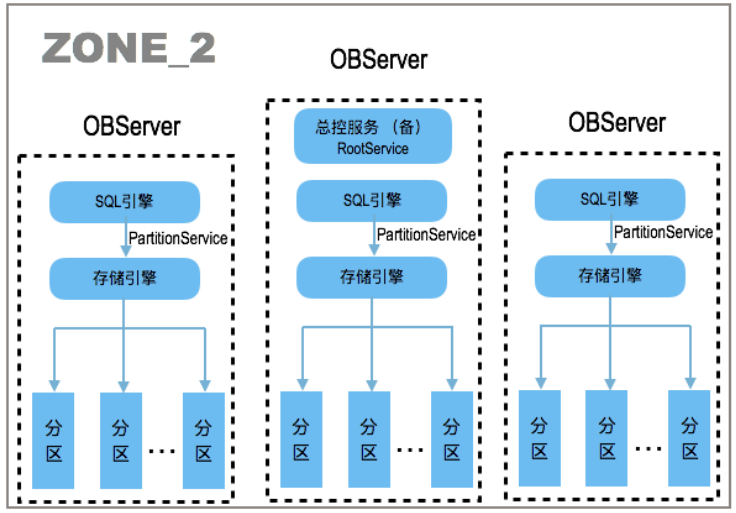
2016年11月16日，阿里巴巴CEO张勇在第三届世界互联网大会上介绍OceanBase数据库

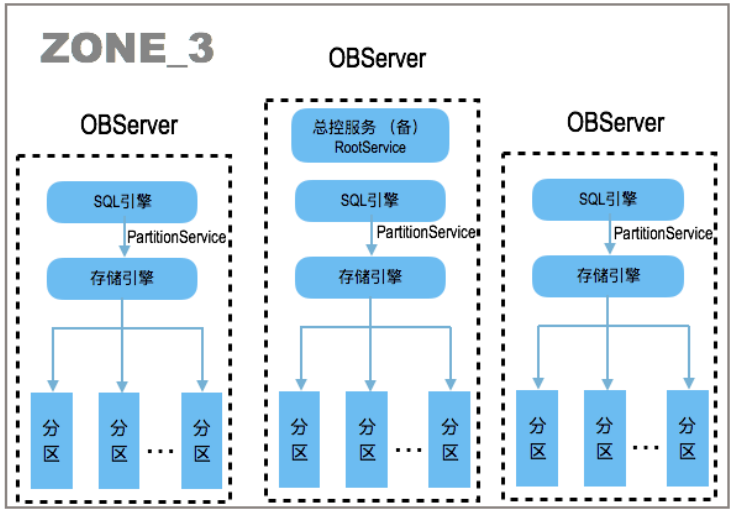
OceanBase
A Powerful Database System

OceanBase架构——集群拓扑

- **多副本**
  - 一般部署为三个子集群(Zone)
  - 每个子集群由多台机器组成
- **全对等节点**
  - 每个节点功能对等
  - 各自管理不同的数据分区
- **不依赖共享存储**



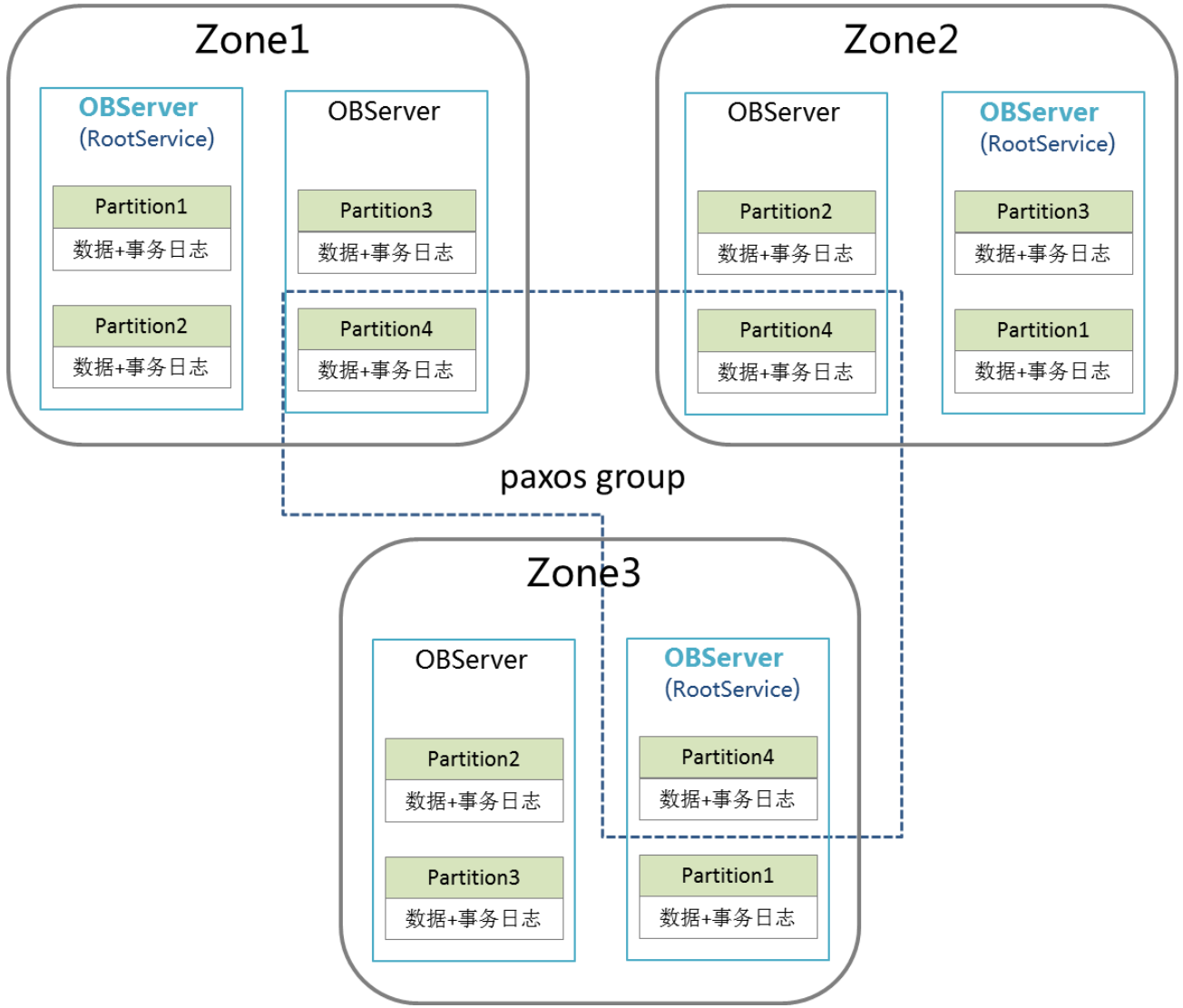




OceanBase
A Powerful Database System

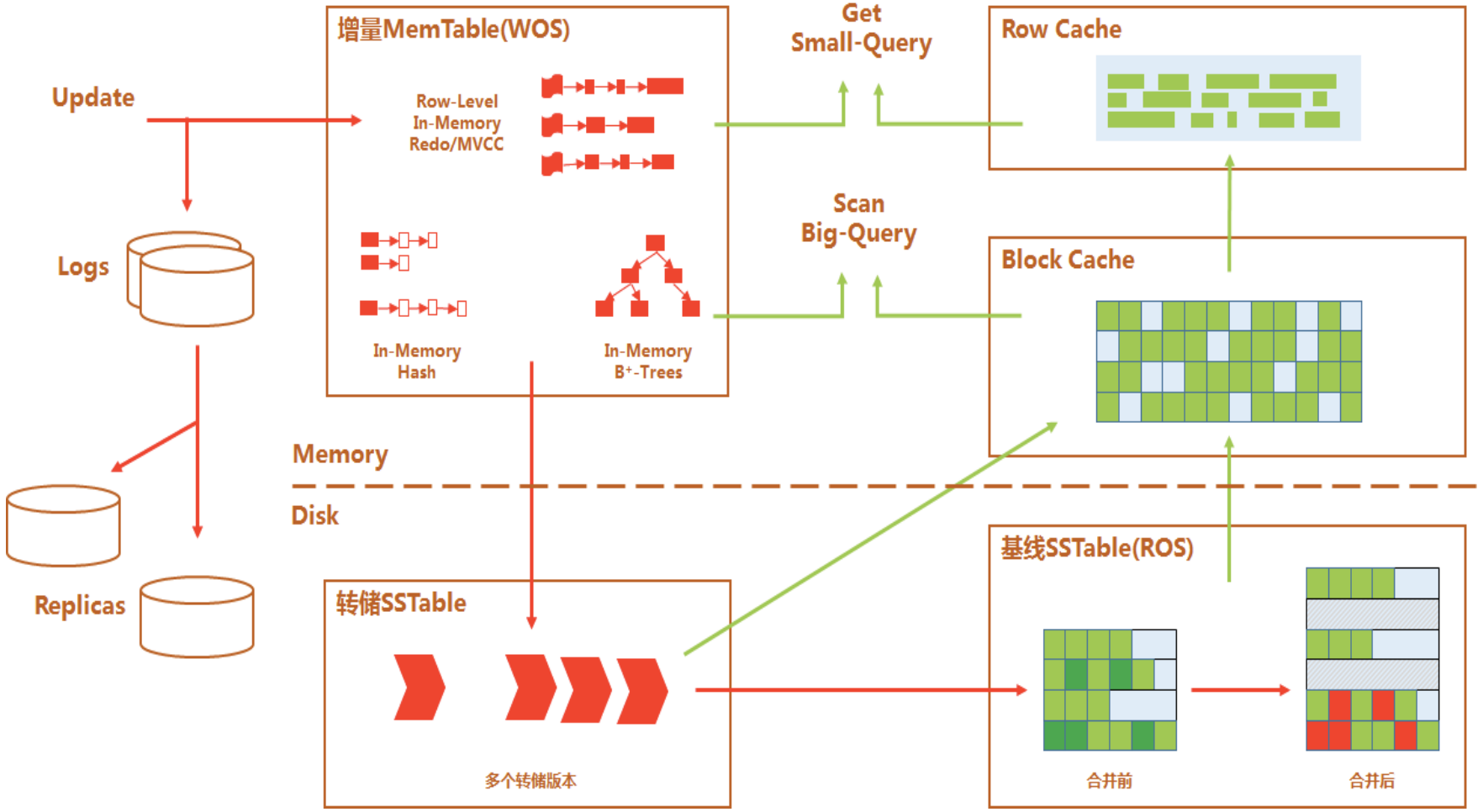
OceanBase架构——分区&分布式协议

- **数据分区**
  - 支持数据分区（partitioning）
  - 各分区独立选主、写日志
- **高可用&强一致**
  - PAXOS协议保证数据（日志）同步到多数机器
  - 故障时自动切主



OceanBase架构——存储引擎

- 动态修改写内存
- 静态数据无修改
- 无随机写，SSD友好
- 批量写，高压缩支持
- 强数据校验

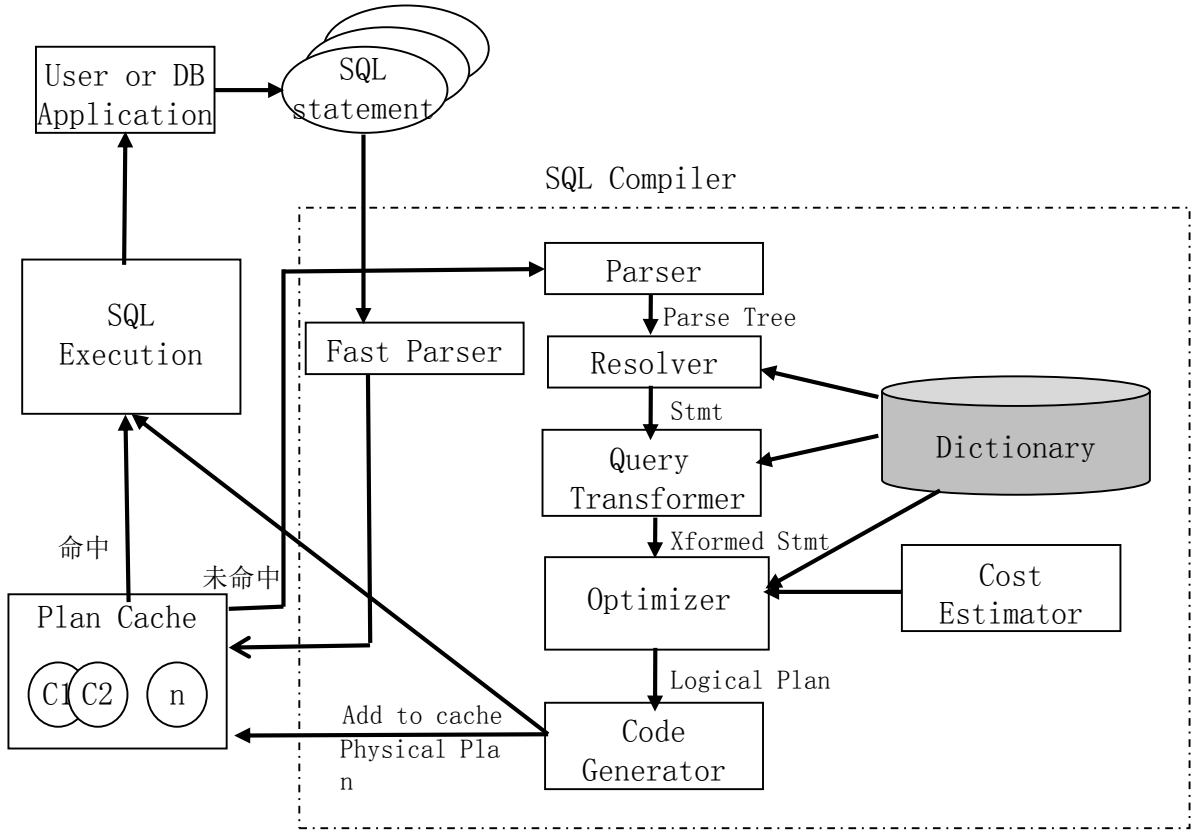


SQL的生命周期

- SQL优化的目标

  最小化SQL执行时间

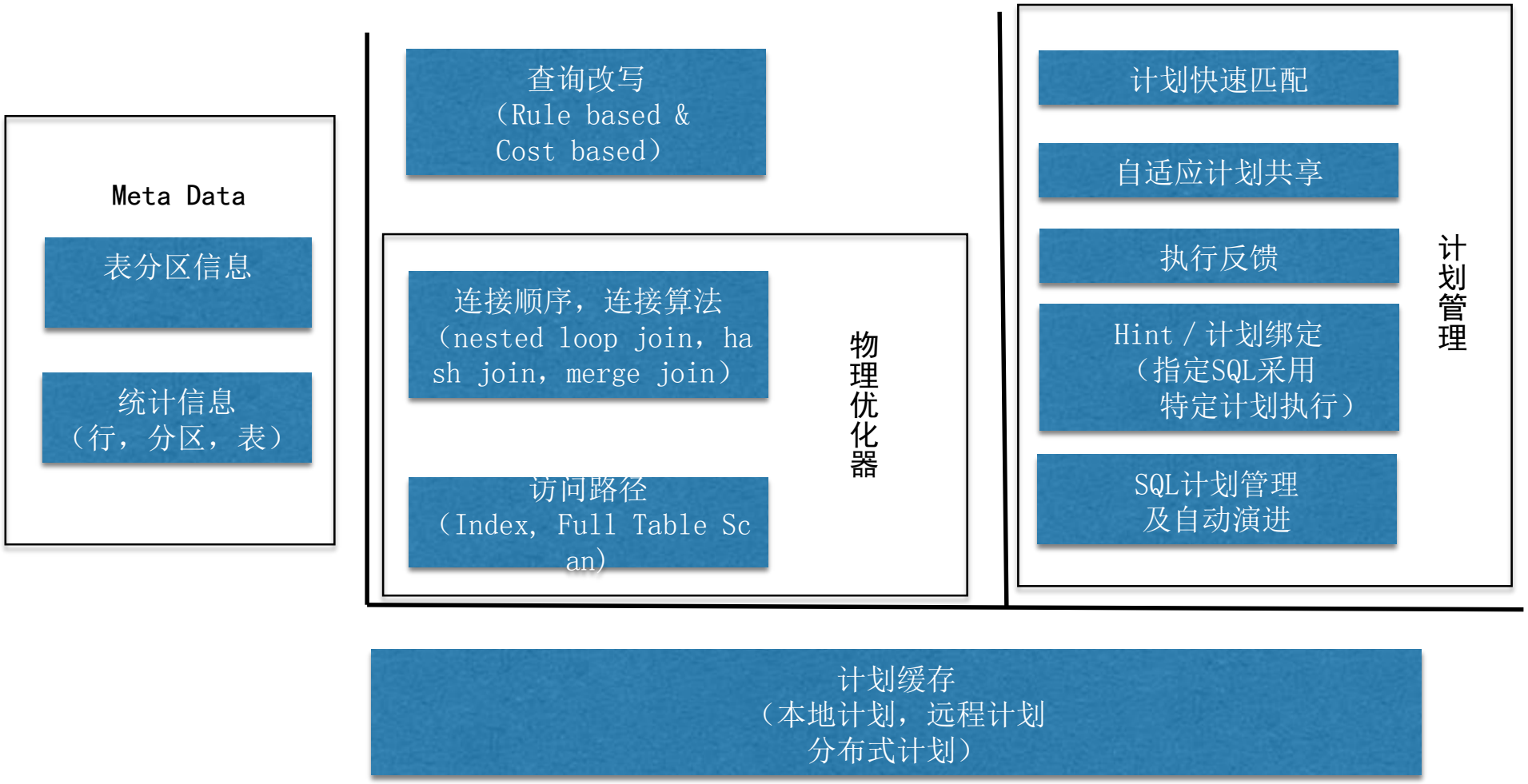
  （计划生成＋计划执行）



基于LSM-Tree结构的代价优化器

- System R模式的基于代价的优化器
- 两阶段优化
  - 第一步：生成基于所有关系都是本地的最优计划（CPU+IO）
  - 第二步：并行执行优化（CPU+IO+Network＋Overhead）
- 代价模型考虑LSM-tree结构的特点
  - MemTable/SSTable 代价采用动态采样计算
  - 分布式share nothing系统
  - 索引回表操作会有额外的代价开销——使用rowkey而不是物理rowid回表速度会慢一些

优化器基本能力


Meta Data
表分区信息
统计信息
（行，分区，表）
查询改写
（Rule based &
Cost based）
连接顺序，连接算法
（nested loop join，ha
sh join，merge join）
访问路径
（Index，Full Table Sc
an）
物理优化器
计划快速匹配
自适应计划共享
执行反馈
Hint / 计划绑定
（指定SQL采用
特定计划执行）
SQL计划管理
及自动演进
计划管理
计划缓存
（本地计划，远程计划
分布式计划）

优化器统计信息

- 表 / 分区 / 行级统计信息

  表：avg rowlen, # of rows

  列：column NDV, null value, histogram，min/max

  分区：分区级别的表统计信息
- 统计在数据进行大版本合并时候进行
- 存储引擎对于某些谓词可以提供较精准的数据量Cardinality估计
  - 谓词可以推出扫描索引的开始和结束区间
  - SStable：每个block都有metadata统计行数
  - MemTable： 关于insert/delete/update操作的metadata
- 动态采样

访问路径

- 选择什么？

  主键扫描 vs 二级索引

  单列索引

  多列索引

- 如何选择？
  - 规则模型
    - 决定性规则（主键覆盖，索引键全覆盖…）
    - 剪枝规则(skyline剪枝规则，多个维度比较）
  - 代价模型
- 考虑因素
  - 扫描范围
  - 是否回表
  - 过滤条件
  - Interesting order

计划缓存

- 计划可以被缓存以备将来使用
- 高效的词法解析器匹配输入查询
- 使用参数化查询优化进行匹配
  - 计划缓存中的每个计划都和一个参数化SQL关联

    参数化SQL: select a, b, c from t1 where a < ? and b < ?;

```
select c1, c2, c3
from t1
where c1 = 1 and c2
like 'senior%'
order by 3 limit 1,
20;
```

add to cache →

```
select c1, c2, c3
from t1
where c1 = @1 and c2
like @2
order by @3 limit @4,
20;

Extra condition:
@3 = 3
```

```
select c1, c2, c3
from t1
where c1 = 2 and c2
like 'child%'
order by 2 limit 1,
20;
```

X *not matched*

{2, 'child%' , 2, 1}

Fast-parser simplify the 'soft-parse' to just s canning for literal values in query

自适应计划共享

- 大小账户问题
- 界定出对bind sensitive的查询
- 执行Feedback机制确定查询bind aware
- 基于存储层信息/histogram获取选择率信息
- 自适应调整Plan Cache中计划的特征值

```
select *
from t1, t2
where t1.c1 = t2.c1 and
t1.c2 = 'salesman';
```

add to cache

```
select *
from t1, t2
where t1.c1 = t2.c1 and
t1.c2 = @1;

@1 selectivity= 10%
```

```
select *
from t1, t2
where t1.c1 = t2.c1 and
t1.c2 = 'president';

 @1 selectivity = 0.1%
```

not matched

自适应计划共享决定不能share，重新生成计划（可能生成同样计划）

Hint/Outline

- 丰富的hint

  access path, join order, join method, parallel distribution…

- Outline绑定

  通过SQL TEXT绑定

```
CREATE OUTLINE otl_idx_c2
ON SELECT/*+ index(t1 idx_c2)*/ * FROM t1 WHERE c2 = 1;
```

  通过SQL ID绑定

```
CREATE OUTLINE otl_idx_c2
ON "ED570339F2C856BA96008A29EDF04C74"
USING HINT /*+ index(t1 idx_c2)*/ ;
```

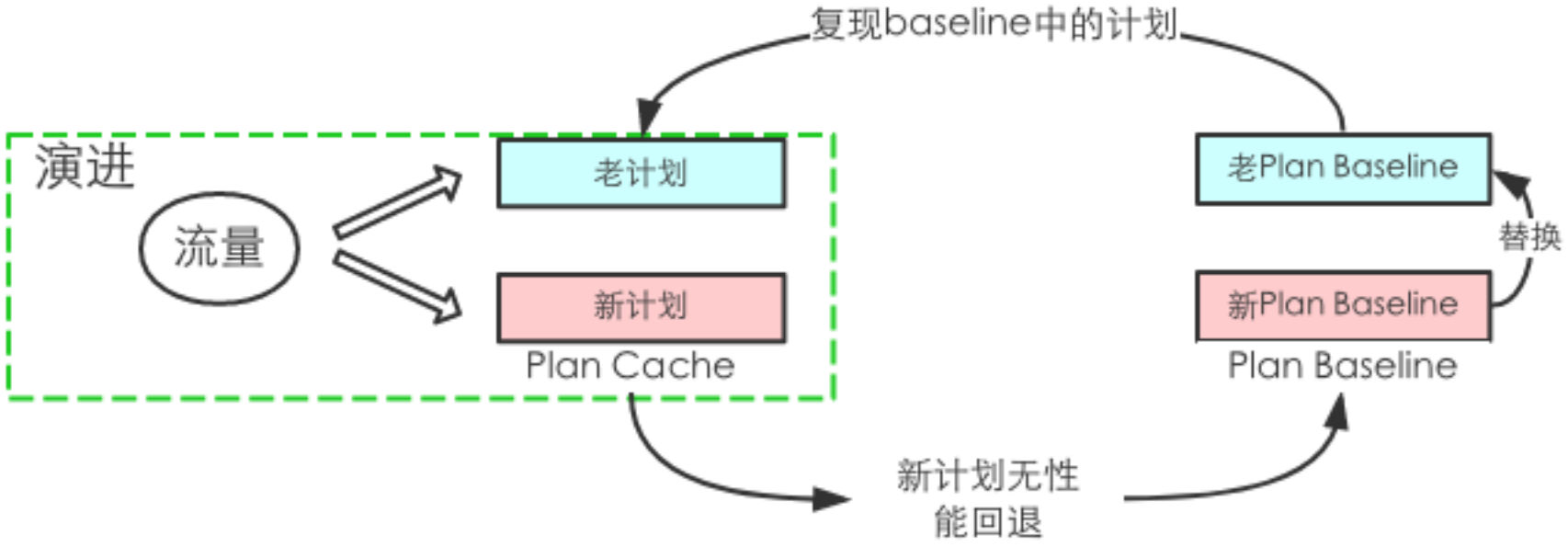
SQL计划管理及演进

- SQL计划管理

  系统升级 / 统计信息更新 / 硬件更新

  计划稳定性

- SQL计划自动演进



分区及分区裁剪

- 多种分区类型

  Range/Hash/Key/List

- 二级分区支持

  Range/Range, Range/Hash, Range/List, Hash/Hash, Hash/Range….

- 静态 / 动态分区裁剪

  inlist， 函数表达式

  join filter

查询改写

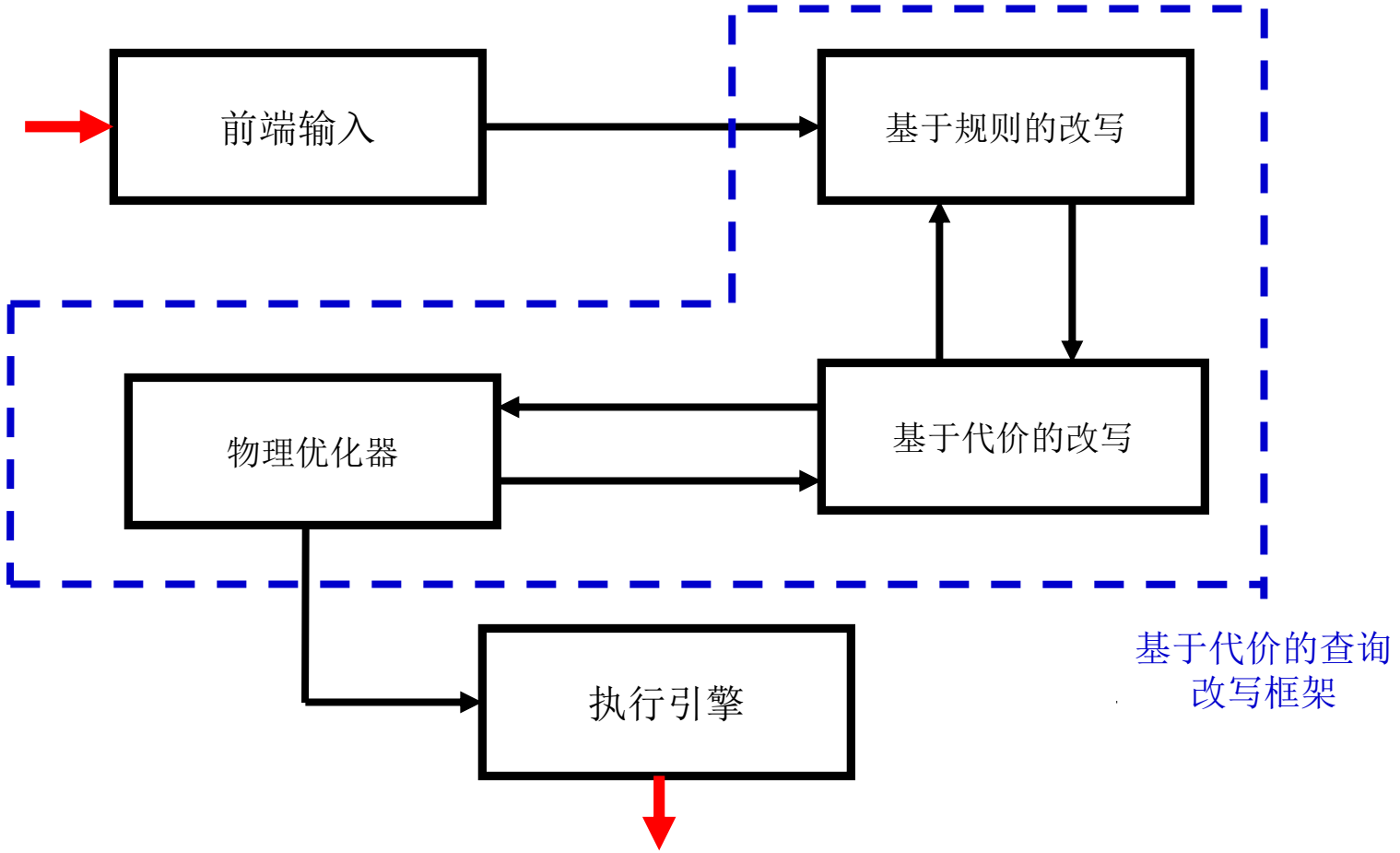
- 基于规则的改写

  视图合并
  子查询展开
  　Anti/Semi Join
  过滤条件下推， 连接条件下推，等价条件推导
  外连接消除
  distinct消除
  sort/order by 消除, sort方法选择
  and more…

- 基于代价的改写

  OR-expansion
  Window function (开发中)
  Group by placement (开发中)
  连接条件下推(JPPD to view/table) (开发中)

基于代价模型的改写框架



子查询展开

```
SELECT c_acctbal
FROM customer
WHERE NOT EXISTS (SELECT *
                  FROM orders
                  WHERE o_custkey = c_custkey);
```

```
SELECT c_acctbal
FROM customer, orders
WHERE c_custkey A= o_custkey;
```

OR Expansion

- 更多访问路径选择

```
select * from t1 where t1.a = 1 or t1.b = 1;
```

```
select * from t1 where t1.a = 1
union all
select * from t1.b = 1 and lnnvl(t1.a = 1);
```

- 更多连接类型选择

```
select * from t1, t2 where t1.a = t2.a or t1.b = t2.b;
```

```
select * from t1, t2 where t1.a = t2.a
union all
select * from t1, t2 where t1.b = t2.b and lnnvl(t1.a = t2.a)
```

Window Function改写

- TPC—H （Q17）

  1G schema，无并行

改写前： 3小时无结果

改写后： 4.9s

```
explain select sum(l_extendedprice) / 7.0 as avg_yearly
from lineitem, part
where p_partkey = l_partkey and p_brand = 'Brand#23'  and p_container = 'MED BOX'
and l_quantity < ( select 0.2 * avg(l_quantity)
                   from lineitem
                   where l_partkey = p_partkey)
```

0	SCALAR GROUP BY			1	459031057
1	EXCHANGE IN DISTR			1	459031057
2	EXCHANGE OUT DISTR			1	459031057
3	MERGE GROUP BY			1	459031057
4	SUBPLAN SCAN		SUBQUERY_TABLE	14652096	456232325
5	WINDOW FUNCTION			43956286	446485188
6	SORT			43956286	438088992
7	HASH JOIN			43956286	262036679
8	JOIN FILTER CREATE			0	385854
9	GRANULE ITERATOR			20513	385854
10	TABLE SCAN		part	20513	385854
11	EXCHANGE IN DISTR			0	42384
12	EXCHANGE OUT DISTR (PKEY)			0	42384
13	JOIN FILTER USE			0	42384
14	GRANULE ITERATOR			654393600	42384
15	TABLE SCAN		lineitem	654393600	42384

```
Outputs & filters:
-------------------------------------
  0 - output([T_FUN_SUM(T_FUN_SUM(SUBQUERY_TABLE.l_extendedprice)) / 7]), filter(nil),
      group(nil), agg_func([T_FUN_SUM(T_FUN_SUM(SUBQUERY_TABLE.l_extendedprice))])
  1 - output([T_FUN_SUM(SUBQUERY_TABLE.l_extendedprice)]), filter(nil)
  2 - output([T_FUN_SUM(SUBQUERY_TABLE.l_extendedprice)]), filter(nil)
  3 - output([T_FUN_SUM(SUBQUERY_TABLE.l_extendedprice)]), filter(nil),
      group(nil), agg_func([T_FUN_SUM(SUBQUERY_TABLE.l_extendedprice)])
  4 - output([SUBQUERY_TABLE.l_extendedprice]), filter([SUBQUERY_TABLE.l_quantity < SUBQUERY_TABLE.win_agg])
      access([SUBQUERY_TABLE.win_agg], [SUBQUERY_TABLE.l_extendedprice], [SUBQUERY_TABLE.l_quantity])
  5 - output([0.2 * T_FUN_SUM(lineitem.l_quantity) / T_FUN_COUNT(lineitem.l_quantity)], [lineitem.l_extendedprice], [lineitem.l_quantity]), filter(nil),
      win_expr(T_FUN_COUNT(lineitem.l_quantity)), partition_by([part.p_partkey]), order_by(nil), window_type(RANGE), upper(UNBOUNDED PRECEDING), lower(UNBOUNDED FOLLOWING)
      win_expr(T_FUN_SUM(lineitem.l_quantity)), partition_by([part.p_partkey]), order_by(nil), window_type(RANGE), upper(UNBOUNDED PRECEDING), lower(UNBOUNDED FOLLOWING)
```

并行执行

- RT时间对于RDBMS查询是一个重要指标
  - Map-reduce的执行性能不能满足OLAP需求
  - 高效的执行计划 / 数据流水线非常关键
- 工作线程组的数量是一个资源使用量，RT和并行度的trade off
- 两级调度
- 自适应的子计划调度框架
- 各节点独立的任务切分
- 容错比传统RDBMS更加复杂

并行执行的概念

- 分治
  - 垂直分治

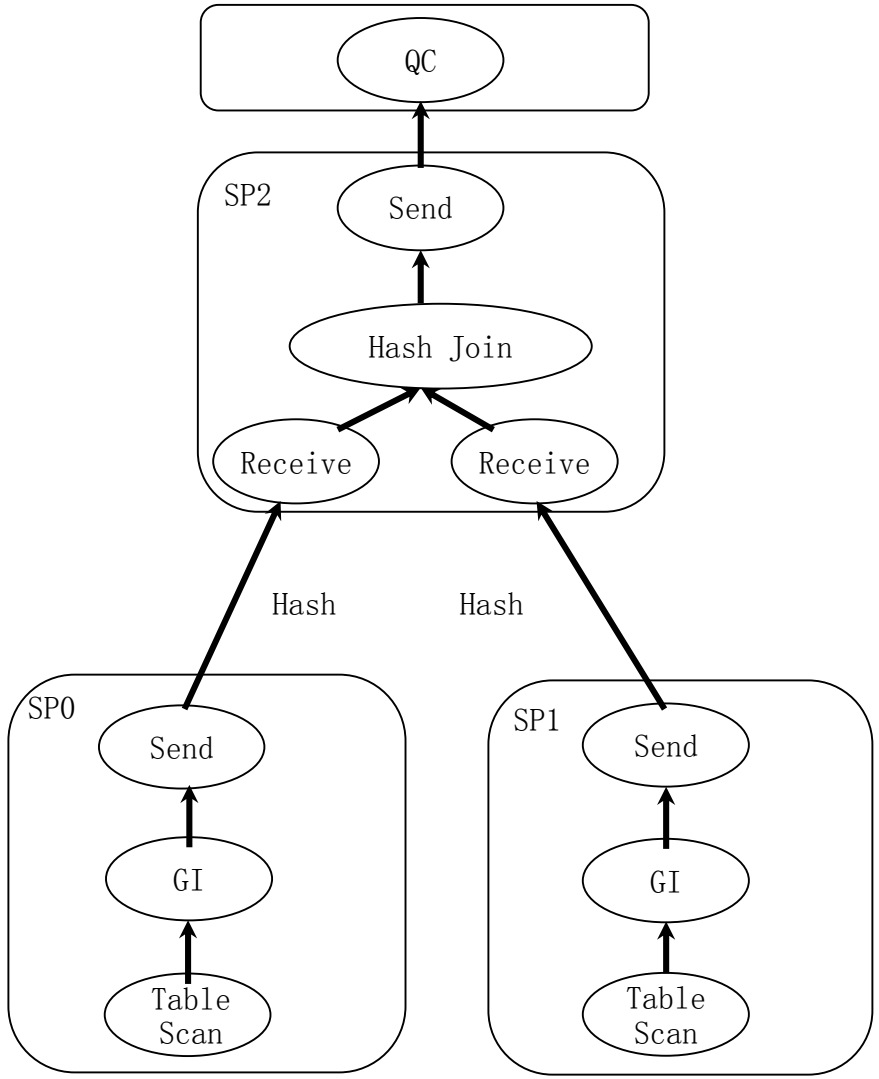
    拆分计划为子计划单元(SubPlan)
  - 水平分治

    GI(Granule Iterator)获取扫描任务
- 数据重分布

  Hash, Broadcast/Replicate,

  Round Robin, Merge Sort,

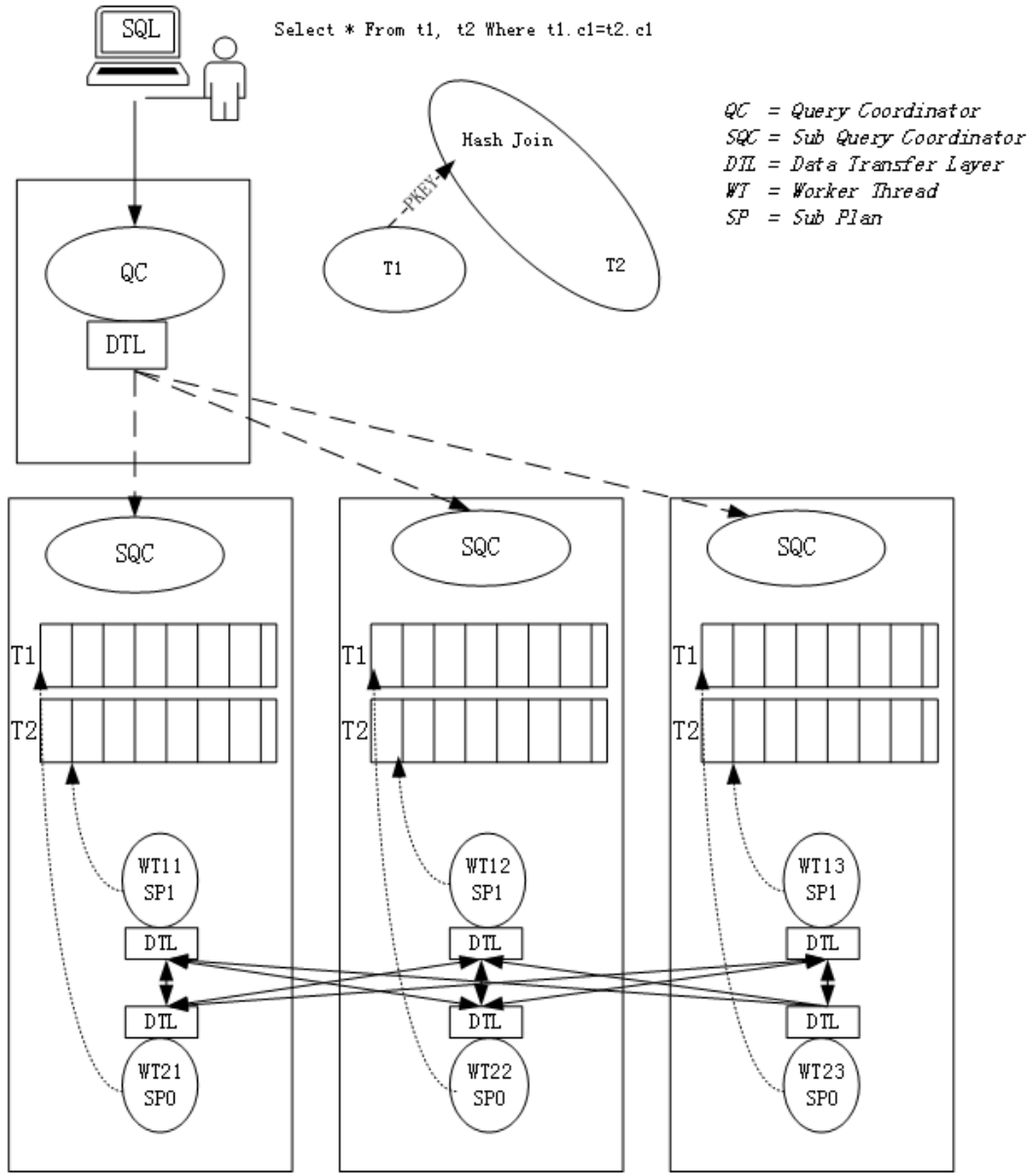
  PKEY(for partial partition-wise join)
- 数据流式传输层（DTL）



两级调度

- 主控节点进行全局调度（QC）
  依据总并行度分配各节点各子计划并行度
- 各执行节点进行本机调度（SQC）

  各节点独立决定水平并行粒度及执行



计划动态调度

- 生产者消费者模型

  构建2组或以上计划子树的执行流水线(垂直并行)

- 根据用户指定或系统资源自适应决定

  在允许的资源使用内，减少中间结果缓存

```
select /*+ parallel(2)*/ * from t1, t2, t3 where t1.c1 =
t2.c1 and t2.c2 = t3.c2;
```

ID	OPERATOR	NAME	EST. ROWS	COST
0	EXCHANGE IN DISTR		1	117
1	EXCHANGE OUT DISTR		1	117
2	HASH JOIN		1	117
3	EXCHANGE IN DISTR		1	75
4	EXCHANGE OUT DISTR (HASH)		1	75
5	MATERIAL		1	75
6	HASH JOIN		1	75
7	EXCHANGE IN DISTR		1	37
8	EXCHANGE OUT DISTR (HASH)		1	36
9	GRANULE ITERATOR		1	36
10	TABLE SCAN	t1	1	36
11	EXCHANGE IN DISTR		1	37
12	EXCHANGE OUT DISTR (HASH)		1	36
13	GRANULE ITERATOR		1	36
14	TABLE SCAN	t2	1	36
15	EXCHANGE IN DISTR		9	39
16	EXCHANGE OUT DISTR (HASH)		9	38
17	GRANULE ITERATOR		9	38
18	TABLE SCAN	t3	9	38

SP4

SP2

SP0

SP1

SP3

并行执行计划

- 支持所有主要算子的并行执行

  nested-loop join, merge join, hash join,

  aggregation, distinct, group by,

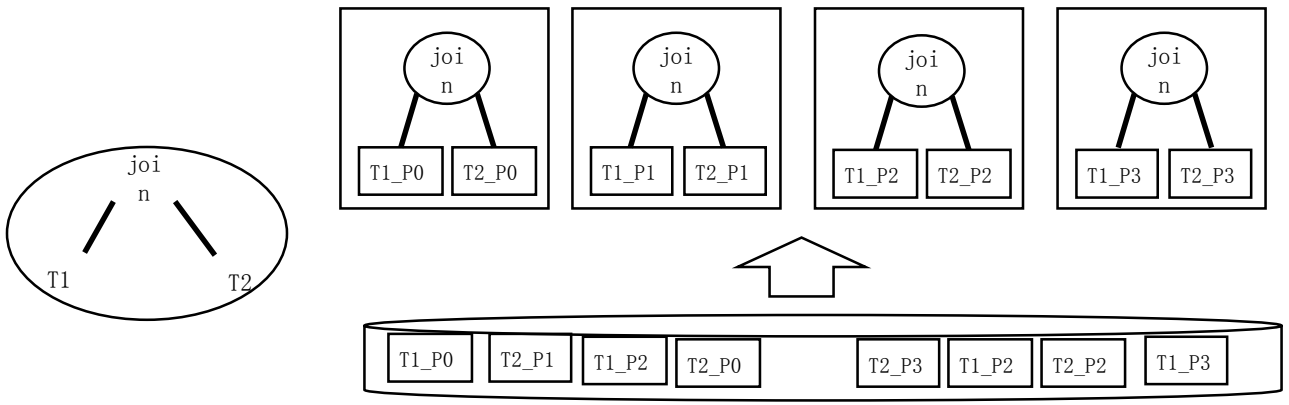
  window function, count, limit…

- 支持丰富的重分布方法和多种候选计划选优

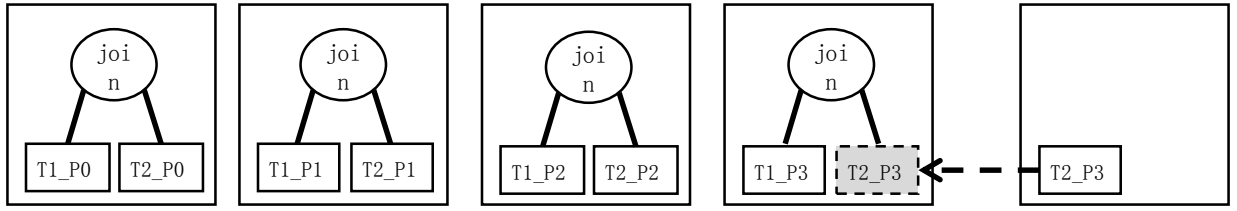
  partition-wise join, partial partition-wise join,

  broadcast, hash, sort(for distributed order by)

- 并行查询的优化技术在MPP架构下有新的问题
  - 分区连接(partition-wise join)要求各表的分区从逻辑上和物理上都是一样的



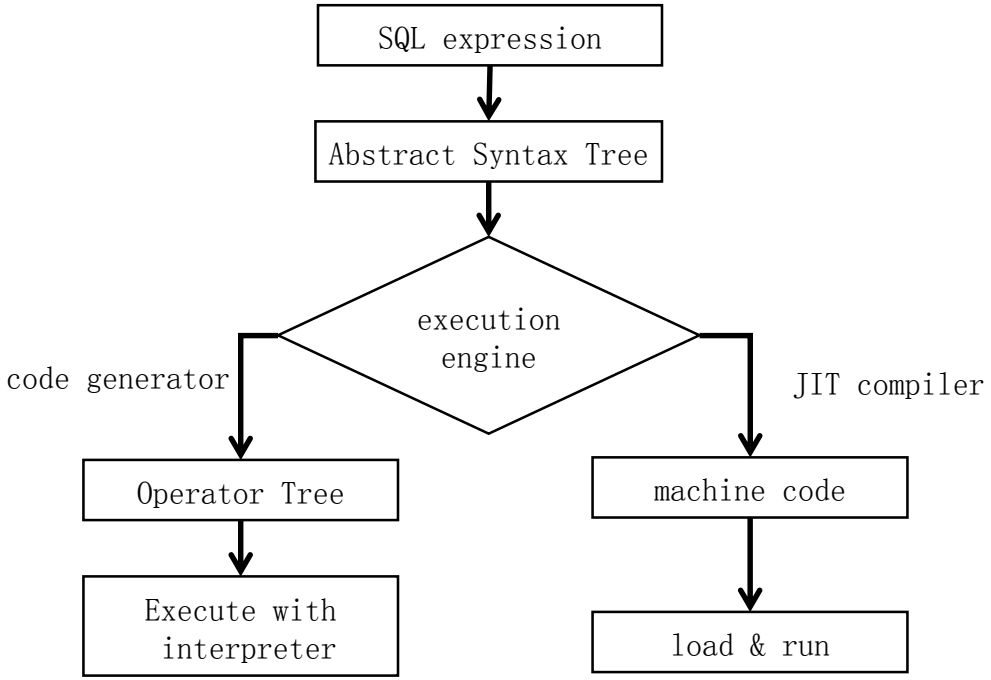
共享磁盘架构下的Partition-wise join只需要两个表按照连接列进行同样的分区即可



MPP share－nothing架构下的Partition-wise join需要两个表按照连接列进行同样分区且对应的分区组都物理分布在相同的机器上

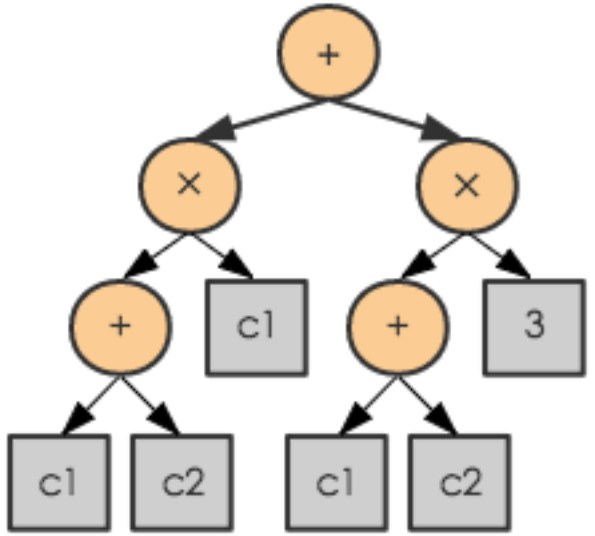
编译执行

- 传统执行方式的问题
  - 类型检查
  - 多态(虚函数)
  - 对于内存操作很低效
- LLVM 动态生成执行码
  - SQL表达式可以支持动态生成执行码(10x 性能提升)
  - 存储过程采用直接编译执行的方式

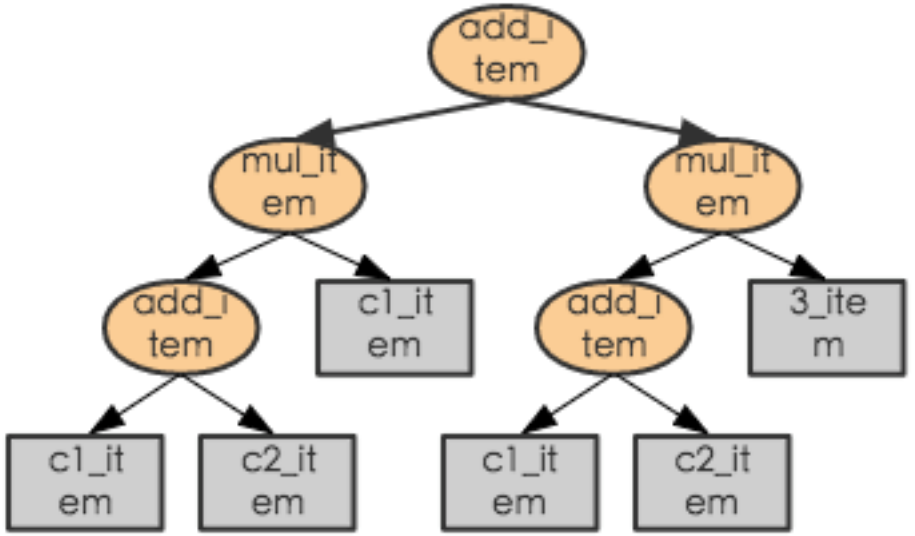


编译执行

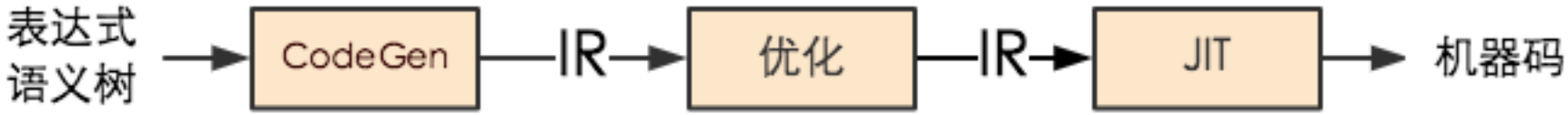
(c1+c2)*c1+(c1+c2)*3



图(a)



图(b)



减少分支判断，类型选择，函数绑定，公共表达式去除

谢 谢 聆 听

OceanBase数据库官网

微信公众号：OceanBase