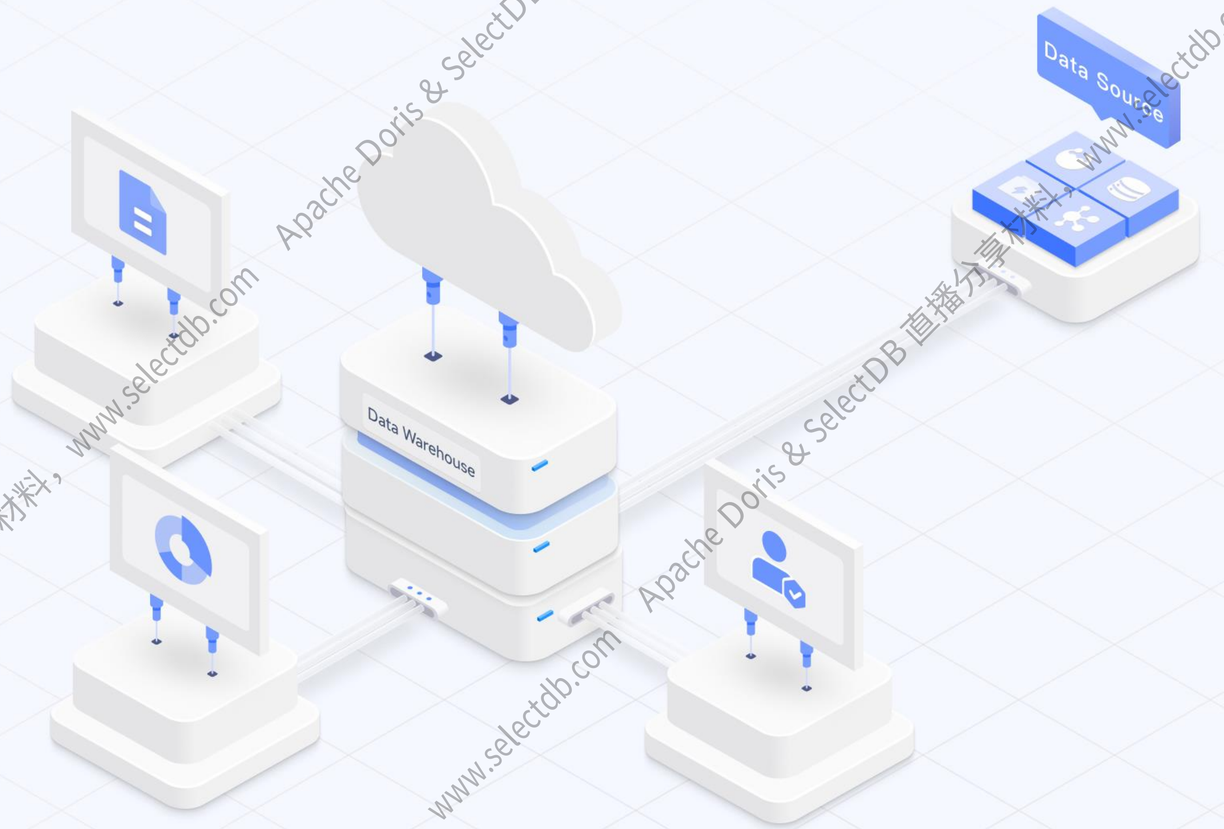
SELECTDB | ACADEMY

Apache Doris & SelectDB

培训与认证课程精选

张敏坷 飞轮科技



目录

CONTENTS

传统数据库 VS MPP 数据库 VS Hadoop

	传统数据库	MPP 数据库	Hadoop 生态
架构	共享资源架构 Shared-Everything	无共享架构 Shared-Nothing	存算分离架构
存储	集中式存储 单节点存储所有数据	分布式存储 每个节点存储部分数据	独立分布式存储 通过网络读取数据
计算	单节点计算 依赖单节点的 CPU/内存	并行计算 每个节点计算子任务后汇总	按需弹性计算 数据从远端读取
扩展性	垂直扩展 （提高单机性能上限）	线性扩展 （节点越多，能力越强）	水平扩展 （存储/计算单独扩展）
性能	小规模数据性能优异 受限于单体硬件	大规模并行计算优异	大规模批处理计算优异 网络 I/O 开销大
成本	硬件升级成本 后期重构成本高	初期硬件成本高	初期硬件成本高 多组件运维成本高

传统数据库 VS MPP 数据库 VS Hadoop

MPP 数据库 VS 传统数据库

优势:

- 大规模计算性能更强
- 扩展性更强

劣势:

- 小数据量性能没有优势
- 不适用事务型的场景
- 存在明显的“木桶效应”

数据分布是决定系统性能、并行效率和扩展性的核心因素

MPP 数据库 VS Hadoop 生态

优势:

- 计算本地化带来更高的性能
- 运维更简单

劣势:

- 扩展不够弹性

通过存算分离架构解决

数据分布对 MPP 数据库的影响

木桶效应

数据倾斜会造成某个子任务拖慢整体任务的进度；

从而无法发挥并行计算的优势。

数据 Shuffle

如果两个节点数据分布一致，本地计算即可；

如果数据分布与查询模式不匹配，导致数据跨节点传输。

忙闲不均

静态负载：部分节点数据量远超其他节点，更容易成为瓶颈；

动态负载：存在数据热点，导致集群整体资源利用率低。

高负载节点硬件损耗更快

扩展失效

扩展节点后，仍然存在数据倾斜问题，添加硬件后，并没有得到性能提升

建表决定了数据的分布

```
CREATE TABLE demo.test_selectdb(
    date_id      DATE,
    site_id      INT          DEFAULT '10',
    city_code    SMALLINT,
    user_name    VARCHAR(32)  DEFAULT '',
    pv           BIGINT       DEFAULT '0'
)
```

字段定义

```
DUPLICATE KEY(date_id, site_id, city_code)
```

模型定义

```
PARTITION BY RANGE(date_id)(
    PARTITION p20200321 VALUES LESS THAN ("2020-03-22"),
    PARTITION p20200322 VALUES LESS THAN ("2020-03-23"),
    PARTITION p20200323 VALUES LESS THAN ("2020-03-24"),
    PARTITION p20200324 VALUES LESS THAN ("2020-03-25")
)
```

分区定义

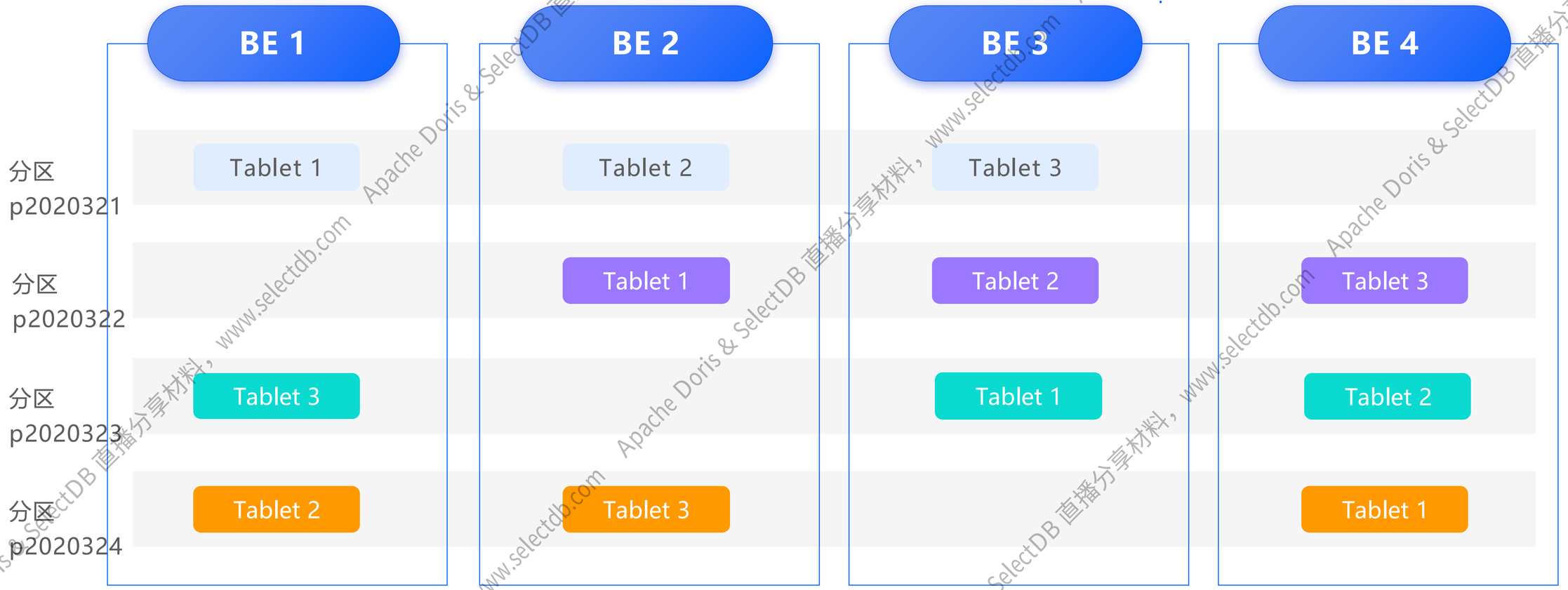
```
DISTRIBUTED BY HASH(site_id) BUCKETS 10;
```

分桶定义

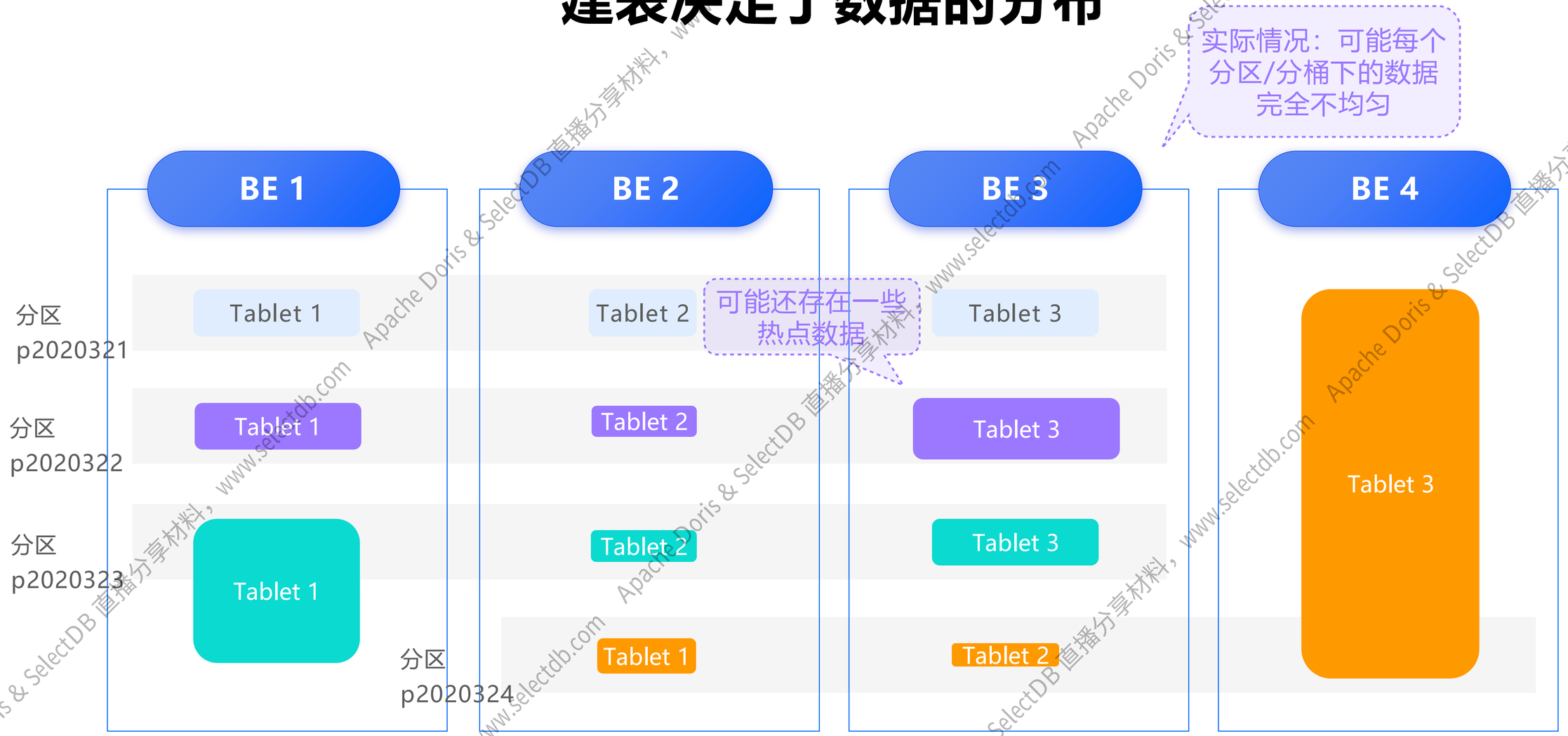
这两个决定数据分布

建表决定了数据的分布

理想状态下，每个 BE 节点均匀的数据分片和数据量



建表决定了数据的分布



通过建表语句影响数据分布

分区策略

管理数据生命周期：

- 通过冷热分离策略，归档历史数据
- 配置过期删除策略，删除失效数据

按业务维度划分数据，减少扫描的数据量

作为分桶策略的基础

分桶策略

保证数据分布：

- 使用高基数字段作为分桶列确保数据均匀分布
- 分桶个数尽量是磁盘和 BE 节点的整数倍

加速查询：

- 分桶字段是过滤条件，减少数据扫描
- 分桶字段是连接条件，减少数据传输

确保每个分桶数据量合适：

- 分桶数据太小，计算任务的启动开销大于计算开销
- 分桶数据太大，无法发挥并行计算优势

目录

CONTENTS

数据组织方式和存储格式

	B/B+ 树	哈希表	合并树	跳表
等值查询	通过主键快速查找 *O(log n)*	通过主键快速定位 *O(1)*	需要在多个节点里查找，合并结果 *O(log n)~O(n)*	通过多层索引快速定位 *O(log n)~O(n)*
范围查询	通过主键快速查找 *O(log n)+k*	全表扫描 *O(n)*	需要在多个节点里查找，合并结果 *O(log n)~O(n)*	通过多层索引快速定位 *O(log n)+k~O(n)+k*
插入	通过主键快速查找，可能触发节点分裂，存在写放大 *O(log n)*	通过哈希函数快速定位到桶，冲突时存在额外开销 *O(1)~O(n)*	直接追加写文件，后续通过compaction减少文件数量 *O(1)*	通过多层索引快速定位添加节点 *O(log n)~O(n)*
更新	通过主键快速查找并更新 *O(log n)*	通过主键快速定位更新 *O(1)*	直接追加写入文件，后续通过compaction操作更新数据 *O(1)*	通过多层索引快速定位并修改值 *O(log n)~O(n)*
删除	通过主键快速删除，或触发节点合并，存在写放大 *O(log n)*	通过主键快速定位 *O(1)*	通常使用标记删除法，后续通过compaction操作实现物理删除 *O(1)*	通过多层索引快速定位删除并修改指针 *O(log n)~O(n)*
适用场景	读写平衡，提供稳定性能	极快的等值查询和写入	优化了写入速度，但查询和compaction 可能成为瓶颈	通常在内存中实现，作为平衡树的平替

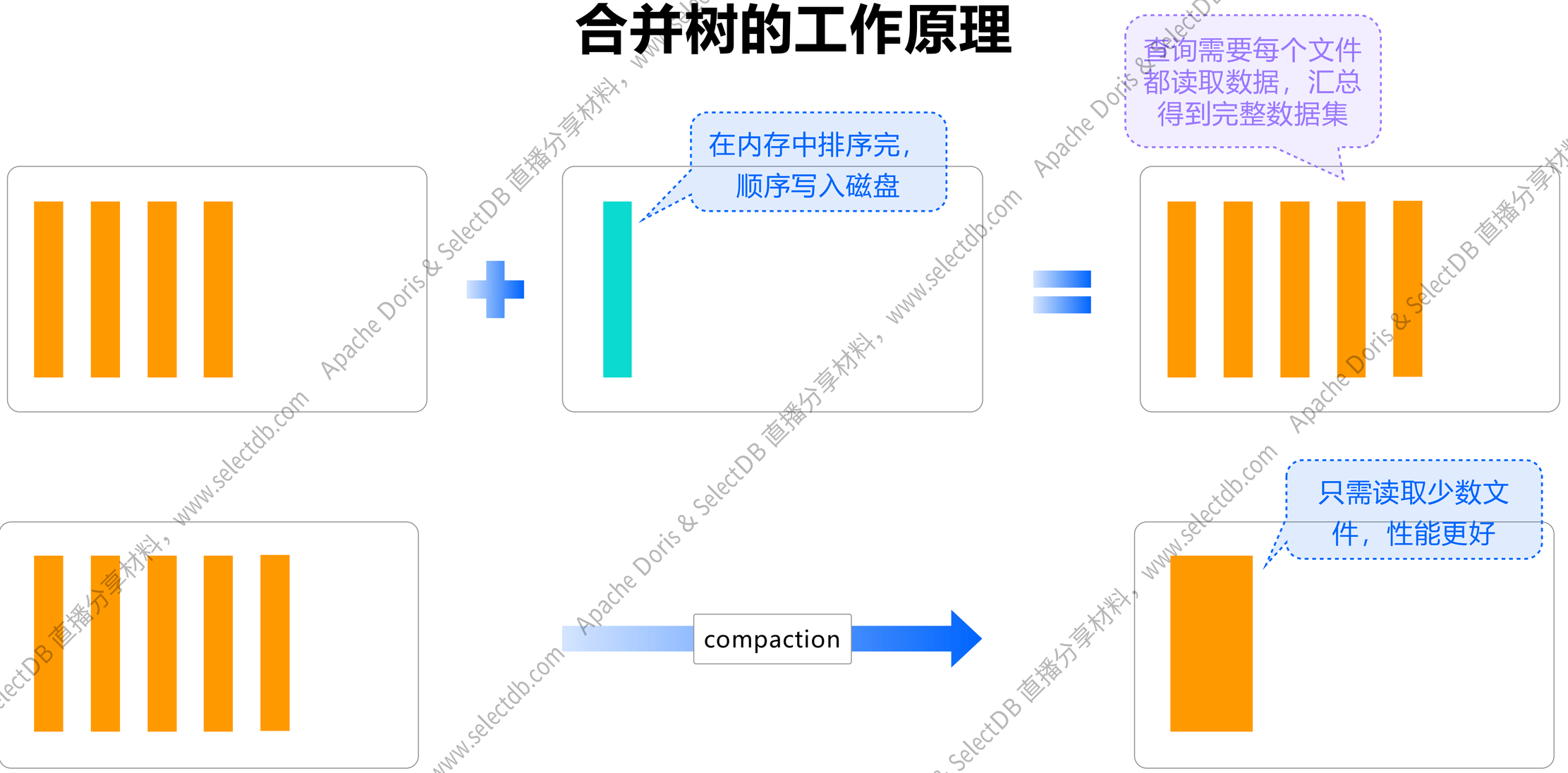
数据组织方式和存储格式

	行式存储	列式存储
存储单位	一行的所有字段连续存储，存完一行再存下一行	同一列的所有值连续存储，存完一列再存下一列
明细查询	整行查询效率高，部分列查询浪费 I/O	部分列查询效率高，整行查询浪费 I/O
聚合查询	需要整行读取后，再聚合，效率低且浪费 I/O	按需读取列进行聚合，可利用向量化技术，效率高
插入	整行插入效率高，部分列插入浪费 I/O	部分列插入效率高，整行插入浪费 I/O
更新	整行更新效率高，部分列查询浪费 I/O	部分列更新效率高，整行更新浪费 I/O
删除	整行删除效率高，部分列删除需要扫描全表	整行删除效率低，部分列删除可通过偏移量快速删除
压缩效率	一行内字段类型多样，重复值少，压缩算法效果差	同列数据类型一致，重复值多，适合字典编码、差值编码等，压缩率通常是行式的 5-10 倍
适用场景	频繁增删改查完整记录	对大量数据进行聚合分析

数据组织方式和存储格式

B/B+树	+	行式存储	=	读写性能稳定 整行增删改查效率高	→ 关系型数据库
B/B+树	+	列式存储	=	读写性能稳定 聚合分析效率高	→ ?
合并树	+	行式存储	=	写入性能极好 整行增删改查效率高	→ 时序数据库 KV数据库
合并树	+	列式存储	=	写入性能极好 聚合分析效率高	→ 分析型数据库

合并树的工作原理



数据操作

插入

导入时的数据量需要比较大，充分发挥磁盘顺序写的优势；

导入的频率不能太快，**防止产生太多小文件**，频繁触发 compaction；

删除

删除操作会转换成导入操作，使用标记删除法，将删除位设为 true；

只有触发 compaction 才会真正物理删除数据，**频繁的删除会导致查询性能急剧下降**；

更新

更新操作则会转换成删除+导入；（目前仅 Unique 模型支持更新）

目录

CONTENTS

Join 操作

数据访问与 I/O 开销

若关联列无索引或其他剪枝优化，可能会触发全表扫描，对于行式存储，可能会读取无用的列

中间结果的存储与传输

Join 过程产生的临时结果集可能存储在内存或磁盘上，分布式系统的 Join 通常伴随着跨节点数据传输

数据匹配的计算开销

Join 操作的实现通常有三种：哈希连接（Hash Join）、嵌套循环连接（Nested Loop Join）、合并连接（Merge Join）

数据访问与 I/O 开销

行式存储

无论如何都需要读取整行数据，在关连列和输出列较少的时候，存在较多的冗余 I/O

列式存储

只需读取关连列和输出列即可，不存在无效 I/O

中间结果的存储与传输

只有分布式系统才有

	无 Shuffle	Broadcast Shuffle	Hash Shuffle	Sort-Merge Shuffle
工作原理	根据两张表的数据分布特征，如相同的分区分桶，无需进行跨节点数据传输	将小表全部传输到大表所在的每一个节点上	将两张表的数据按关连字段哈希后，分别发送到对应节点	将两张表的数据按关连字段哈希和排序，分别发送到对应节点进行归并排序
I/O开销	无网络I/O开销	小表广播的I/O开销	两张表传输的I/O开销	两张表传输排序的I/O开销
数据倾斜	不会产生	不会产生	可能存在	会比 Hash Shuffle好一点
适用场景	两张表数据满足特定分布	大表关连小表	Key分布均匀	超大规模数据量，常见于大数据中的批处理

Colocate Join

Broadcast Join

Bucket Shuffle
Partition Shuffle

数据匹配的计算开销

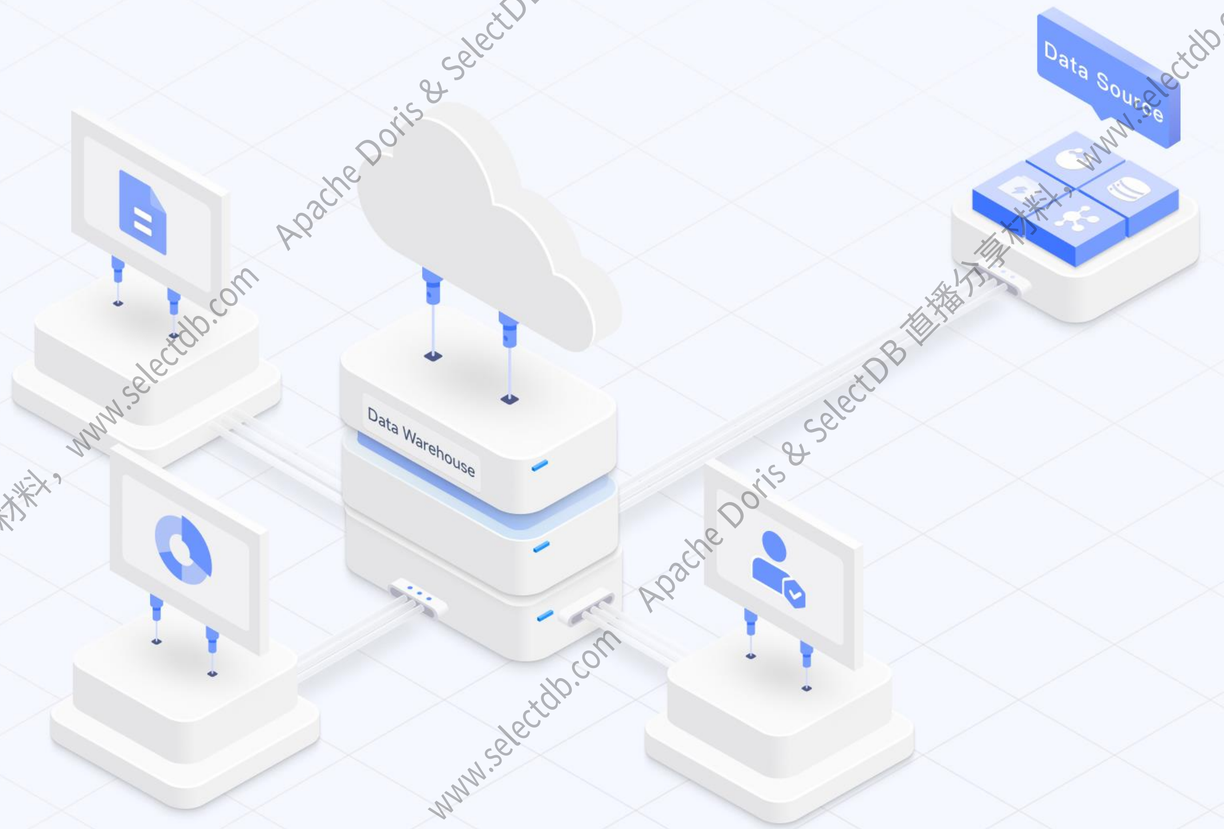
	哈希连接	嵌套循环连接	合并连接
工作原理	较小的表在内存中构建哈希表，顺序扫描较大的表，每一行在哈希表中匹配关连的值	使用双层for循环，比较两张表中的每一条数据是否满足关连条件	将两张表按照连接列排序后，使用双指针扫描匹配关连的值
时间复杂度	$O(M+N)$	$O(M\times N)$	$O(M\times log(M)+N\times log(N))$
并行度	并行性好。可将左表和右表分块，并行构建哈希分表进行探测	并行度差。通常难以切分，或切分需要再汇总	连续存储在排好序之后，并行度好
支持的连接	仅支持等值连接	支持所有连接类型	主要支持等值连接

如何让 Join 更快

- 尽量使用等值连接，关联的列越少越好
- 尽量使用小表驱动大表
- 尽量使用分桶字段作为关联条件
- 有条件可以使用 Colocate Join
- 尽量避免存在倾斜的列作为关连条件

SELECTDB | ACADEMY

Q&A

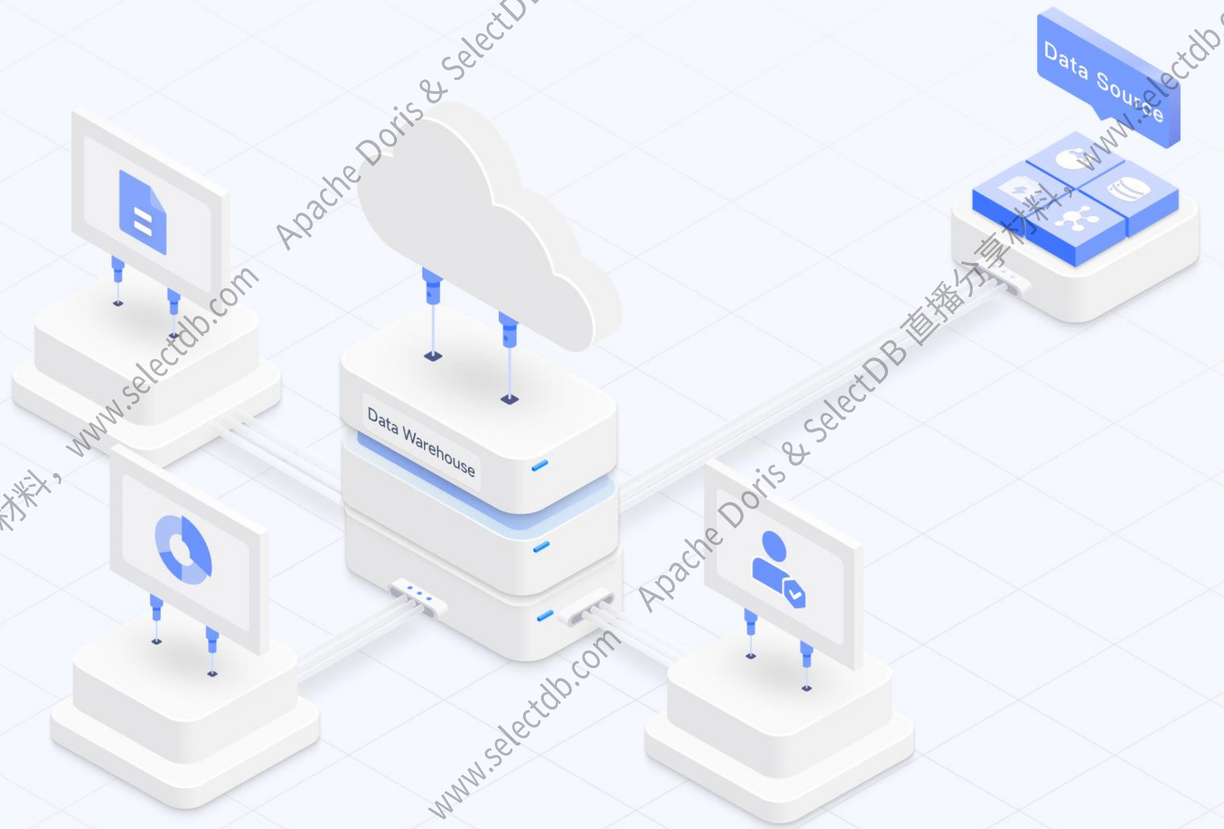


课程官网

加入交流群

SELECTDB | ACADEMY

感谢学习！



课程官网

加入交流群

SELECTDB | ACADEMY

演讲资料扫码进入培训交流群获取

联系我们

www.selectdb.com

400-092-6099

微信公众号

免费试用

在线咨询

培训学习群