内存数据库简述及索引优化

汇报人：周帆

DSC
分布式存储与计算实验室

2022.03.25

目录

01 内存数据库简述

01 内存数据库简述

1.1 基础概念

定义：

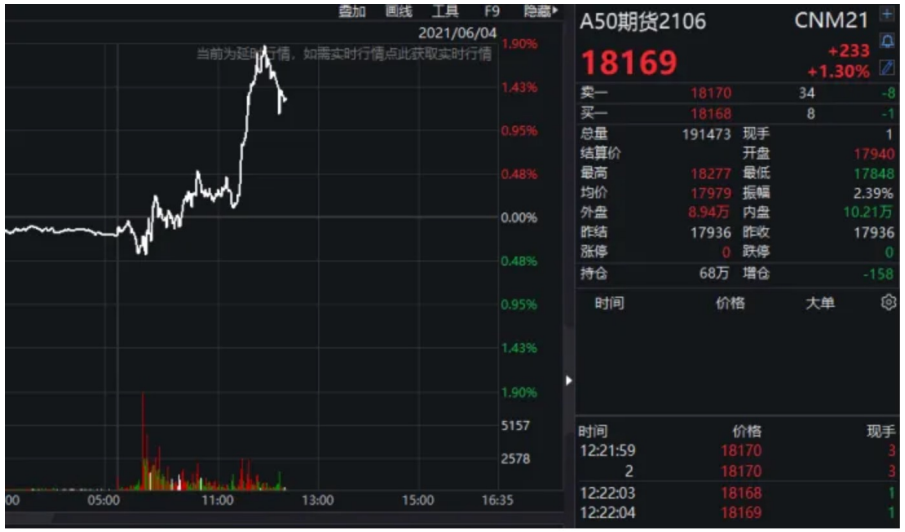
1. 数据全部或大部分放入内存
2. 磁盘作为数据的后备存储设备

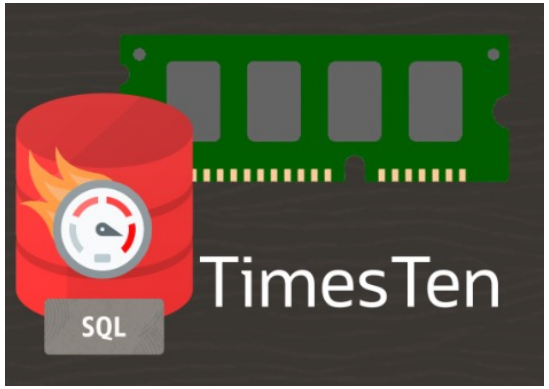
应用：

- 电信行业 — 实时查询
- 证券行业 — 证券交易
- ……

主流内存数据库：

FastDB、Oracle Timesten 、Redis等

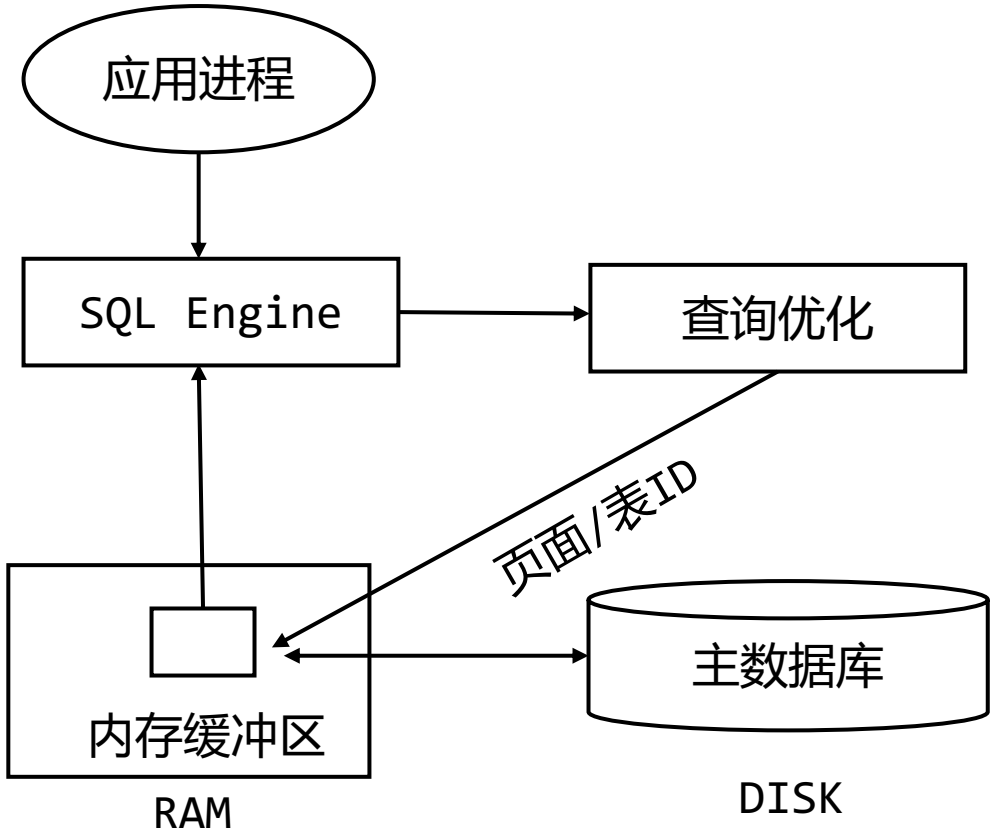




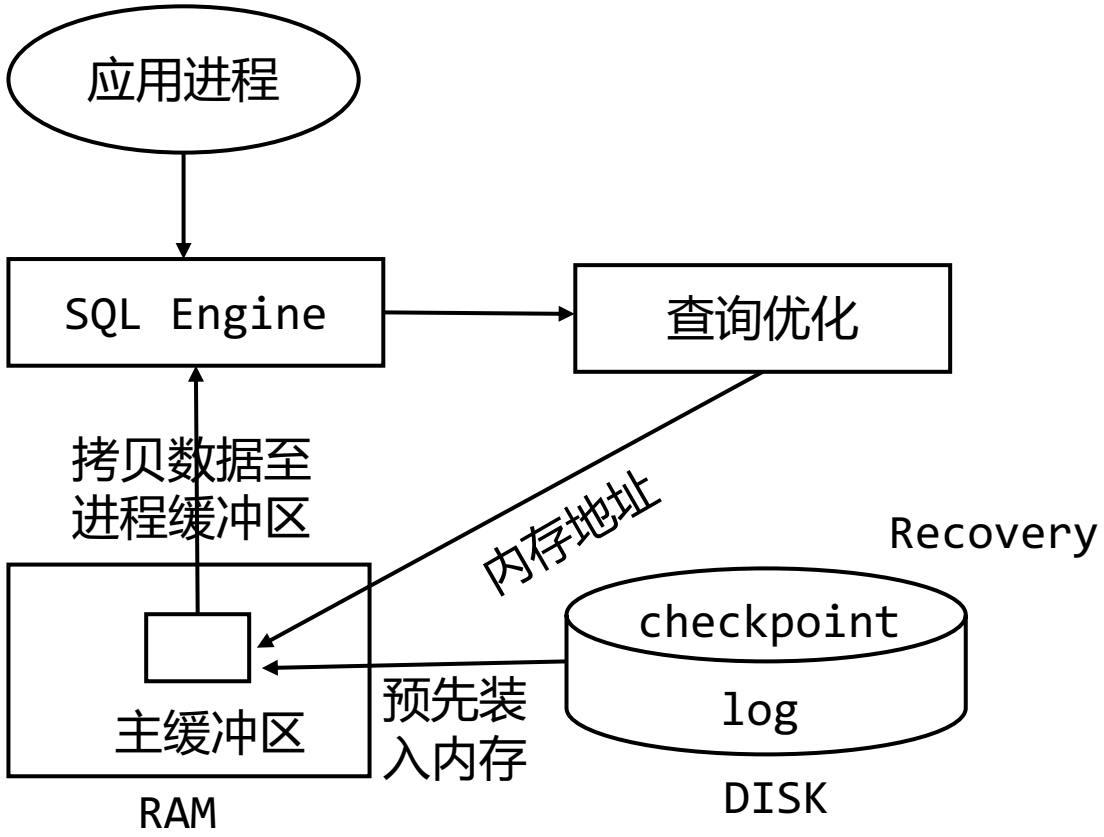
01 内存数据库简述

分布式存储与计算实验室

1.2 DRDB vs MMDB



磁盘数据库



内存数据库

01 内存数据库简述

1.3 特性

1．高吞吐率和低延迟

数据可直接被处理器访问，无需磁盘数据库的缓冲区机制

2．并行处理能力

多通道内存访问机制

3．硬件相关性

内存数据库性能受硬件特性直接影响，主要是多核处理器、高性能存储器、高速连通（数据总线）

01 内存数据库简述

1.4 关键技术

1．数据存储

行存、列存、混合存储等

2．查询处理及优化

查询处理性能主要由内存访问性能和内存处理性能决定

- 内存访问
  - 面向cache的查询技术（造价高，容量小）
  - 索引技术（hash、B+Tree、Trie等）
- 内存处理
  - 面向多核的查询处理技术（单个cpu）
  - 面向众核的查询处理技术（多个cpu）

内存数据库简述

1.4 关键技术

3．并发控制

由于事务执行时间较短，系统冲突较少，可以采用较大的封锁粒度、乐观加锁方式等并发控制方法

4．恢复机制

redo log，undo log等存于磁盘

01 内存数据库简述

1.5 总结

	内存数据库	磁盘数据库
架构	内存为中心	磁盘为中心
缓冲管理	不需要	需要
I/O	很少	多数
响应速度	微秒到毫秒级	毫秒级
数据容量	较少	超大数据量
数据特性	短暂	持久
并发控制	大粒度锁	细粒度锁
查询优化	基于cpu以及内存代价	基于I/O代价

02 / 索引及优化

02 索引及优化

2.1 索引选择

目标：

1. 如何减少内存开销和加快查询时间
2. 如何更有效的使用CPU周期

场景：

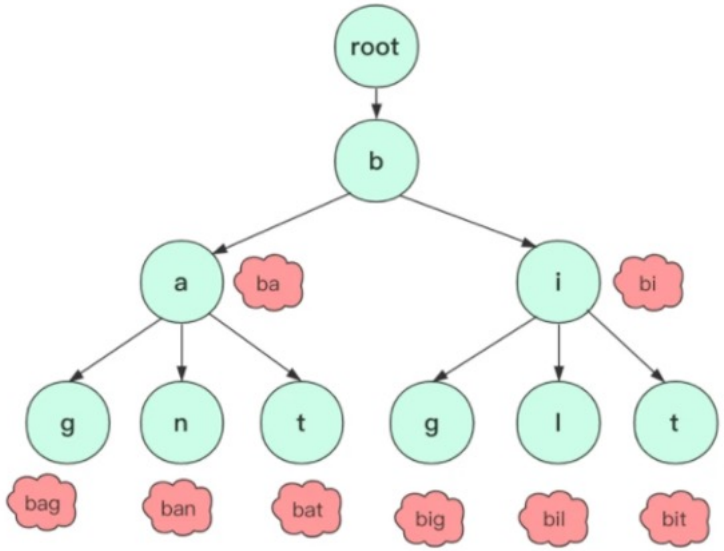
- 搜索引擎 —— Trie，ART，HOT等
- redis —— skiplist
- ……

02 索引及优化

分布式存储与计算实验室

2.2 Trie（字典树、前缀树）

1. 核心思想-空间换时间
2. 基本性质
   I. 相同前缀
   II. 根节点不包含字符（只包含指针），其余每一个节点都只包含一个字符
   III. 连接路径字符，组成字符串（节点不显式存储key）
   IV. 时间复杂度O(m)，m为key的长度



02 索引及优化

2.2 Trie（字典树、前缀树）

3. 实现

I. 数组

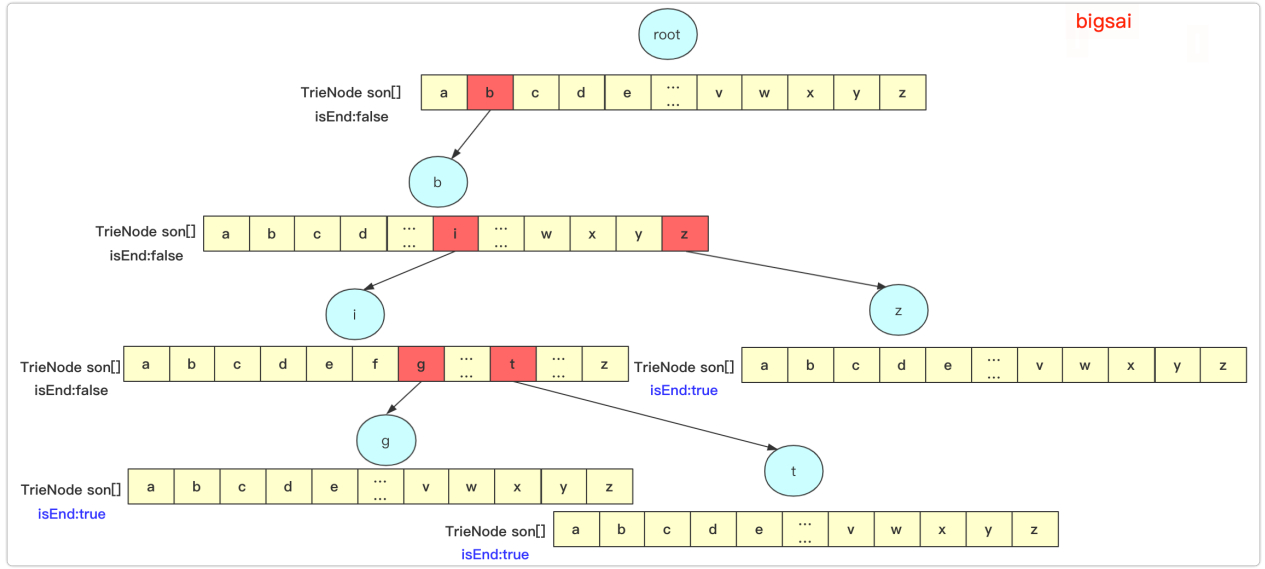
查询快，支持范围查询，空间浪费严重

II. 链表

查询慢，不支持范围查询，节约空间

III. 哈希表

查询快，不支持范围查询，重组代价大



英文字母的字典树是一个26叉树，数字的字典树是一个10叉树

02 索引及优化

2.2 Trie（字典树、前缀树）

4. 问题

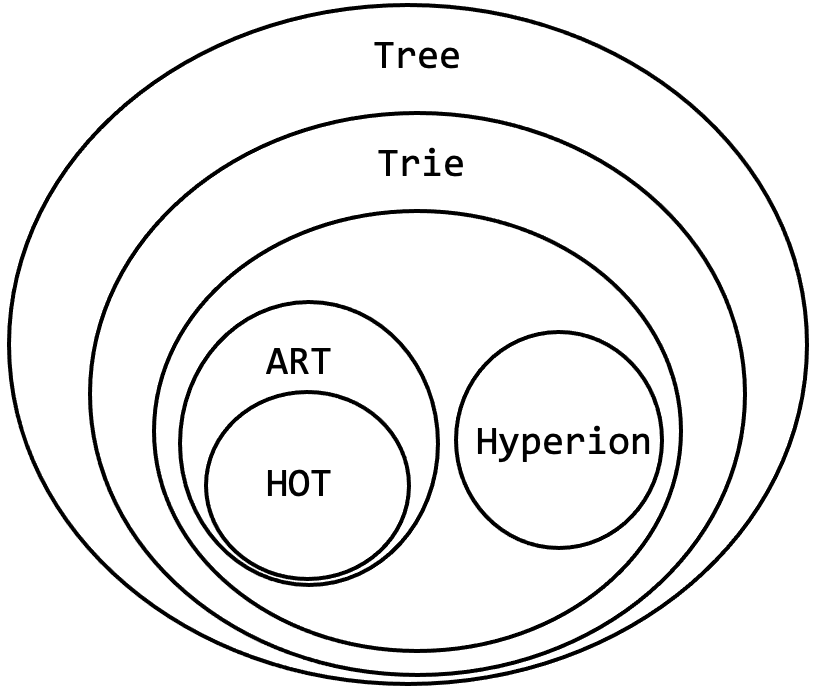
I. 单个字符单独存储导致树的高度较大，时间延迟过高

II. 大量预分配导致空间浪费

5. 优化

I. 加快查询时间 —— 单节点存储多个字符、利用CPU周期

II. 减少内存开销 —— 顺序存储、默认多个节点类型



03 / Adaptive Radix Trie (ART)

02 Adaptive Radix Trie（ART）

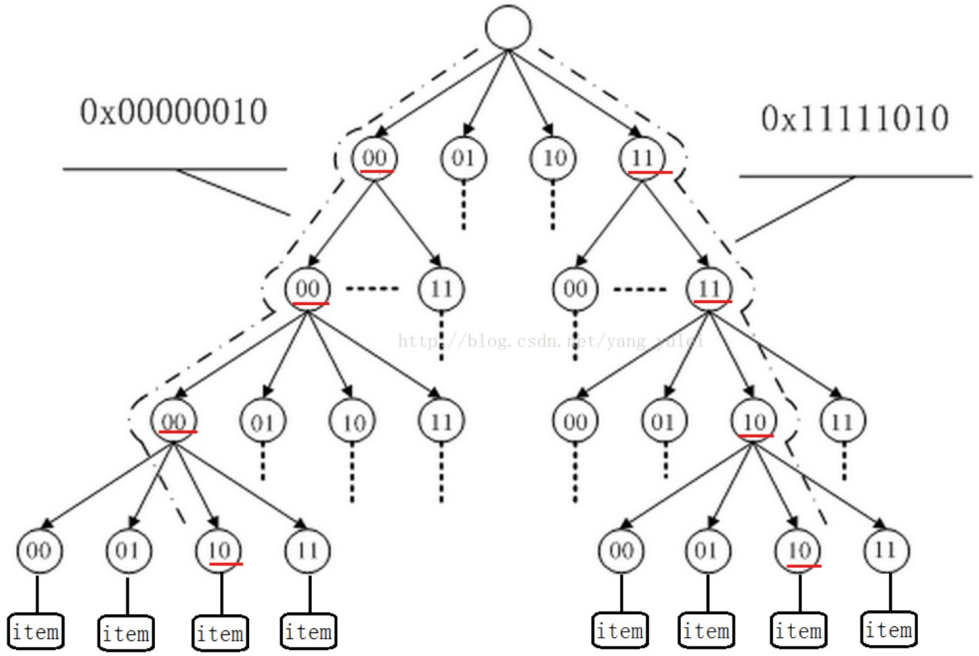
分布式存储与计算实验室

3.1 Radix Trie（基数树）

特点：

1．将所有key用二进制表示

2．多个字符存入一个节点



优势：

1．减少分支数，从而减少无用指针浪费

2．增大共同前缀概率，缩短压缩路径

02 Adaptive Radix Trie (ART)

分布式存储与计算实验室

3.2 Adaptive Radix Trie (自适应基数树)

优化一：压缩路径策略

1. Lazy expansion
   I. 针对leaf node
   II. 当inner node需要区分leaf node才被初始化

2. Path compression
   I. 针对inner node
   II. 合并只有一个child node的所有inner node

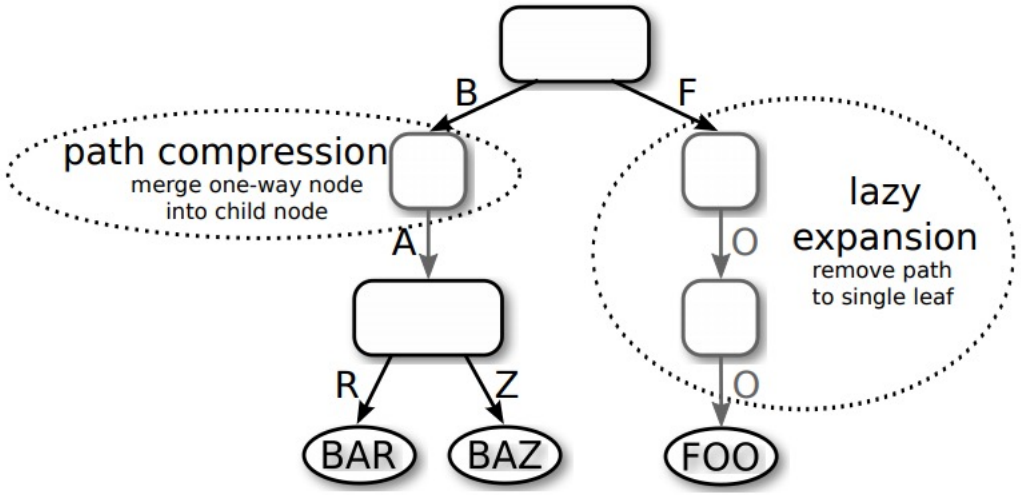


Fig. 6. Illustration of lazy expansion and path compression.

02 Adaptive Radix Trie（ART）

3.2 Adaptive Radix Trie（自适应基数树）

3. 合并节点后，两种处理方式

a）乐观法

比较字符长度，直接跳过，leaf node再比较

优点：对long字符友好

缺点：对short字符不友好，增加比较时间

b）悲观法

不跳过，依次比较字符

优点：对short字符友好

缺点：对long字符不友好，需要依次比较

ART采用混合方法，0～8B，采用悲观法，超过8B，动态转换为乐观法

```
search (node, key, depth)
1  if node==NULL
2    return NULL
3  if isLeaf(node)
4    if leafMatches(node, key, depth)
5      return node
6    return NULL
7  if checkPrefix(node,key,depth)!=node.prefixLen
8    return NULL
9  depth=depth+node.prefixLen
10 next=findChild(node, key[depth])
11 return search(next, key, depth+1)
```

悲观法

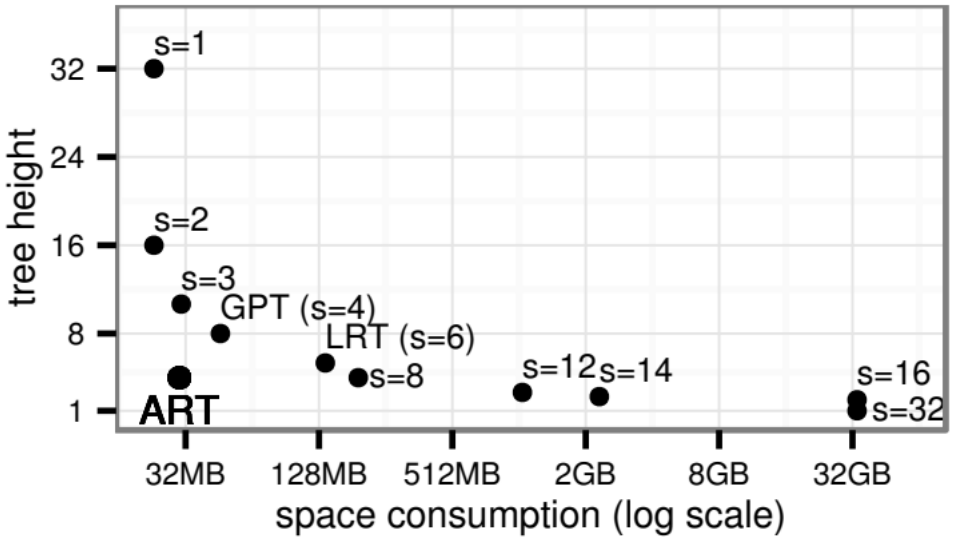
乐观法

02 Adaptive Radix Trie（ART）

3.2 Adaptive Radix Trie（自适应基数树）

优化二：增大span

1. Span（s）参数：二进制位数/单个字符的宽度
2. 对于给定的s，用长度为2^s的指针数组
3. 对于每个字符，ART用八位二进制数表示（s = 8）



4. 平衡二叉树 vs ART

前者每次能排除一半的值，对于s > 1，ART能排除更多的值

02 Adaptive Radix Trie（ART）

3.2 Adaptive Radix Trie（自适应基数树）

优化三：Adaptive Nodes

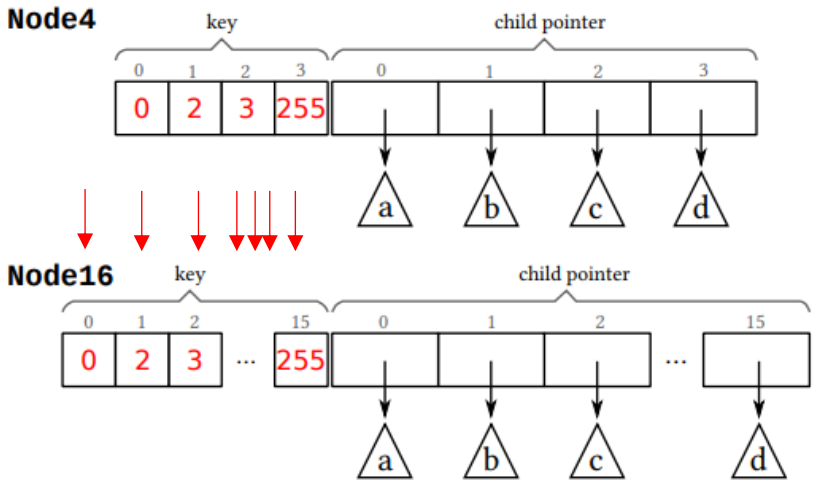
默认四种节点类型

1. Node4

   I. 4个key和4个子节点指针

   II. Key与pointer一一对应

   III. 范围查询 —— 顺序遍历



2. Node16

   I. 16个key和16个子节点指针

   II. 使用SIMD指令，直接把key同最多16个数组值相比较

   III. 范围查询 —— 顺序遍历

02 Adaptive Radix Trie（ART）

分布式存储与计算实验室

3.2 Adaptive Radix Trie（自适应基数树）

优化三：Adaptive Nodes

默认四种节点类型

3. Node48

I. 256个key和48个子节点指针

II. O(1)定位，空间换时间

III. 范围查询 —— O(1)

4. Node256

I. 256个子节点指针

II. O(1)定位

III. 范围查询 —— O(1)

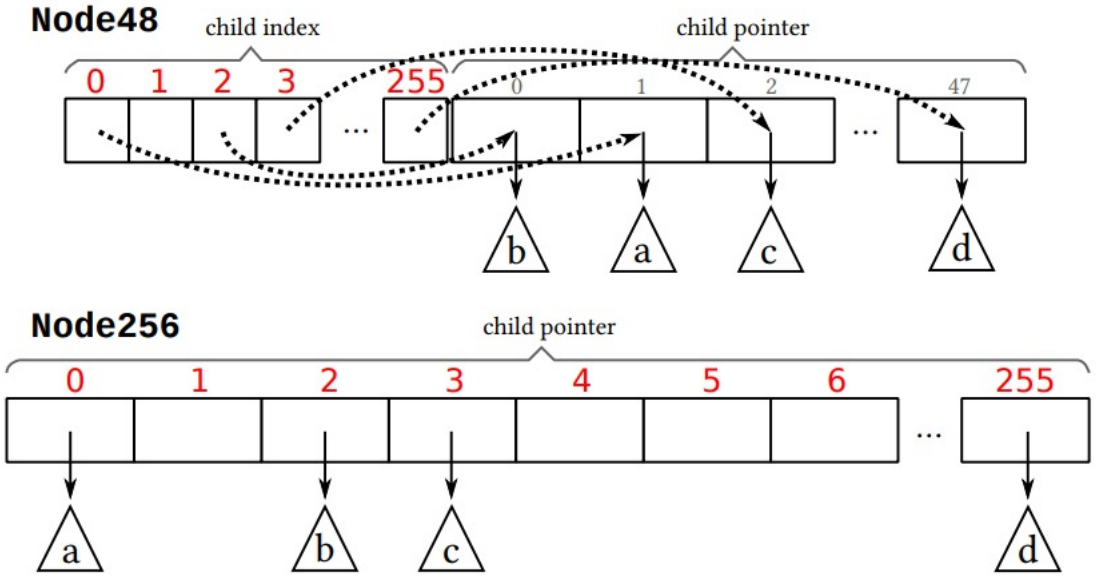


Fig. 5. Data structures for inner nodes. In each case the partial keys 0, 2, 3, and 255 are mapped to the subtrees a, b, c, and d, respectively.

02 Adaptive Radix Trie (ART)

3.2 Adaptive Radix Trie (自适应基数树)

优化三：Adaptive Nodes

空间消耗

TABLE I
SUMMARY OF THE NODE TYPES (16 BYTE HEADER, 64 BIT POINTERS).

Type	Children	Space (bytes)
Node4	2-4	$16 + 4 + 4 \cdot 8 = 52$
Node16	5-16	$16 + 16 + 16 \cdot 8 = 160$
Node48	17-48	$16 + 256 + 48 \cdot 8 = 656$
Node256	49-256	$16 + 256 \cdot 8 = 2064$

02 Adaptive Radix Trie（ART）

3.3 总结

核心思想

增大span并固定，动态选择节点类型

存在问题

I. 树高受数据分布影响

II. 对于sparsely key，用Node256存储，也会导致大量节点空间浪费

思考

能否通过改变span，来降低Radix Trie树高，并减少空间

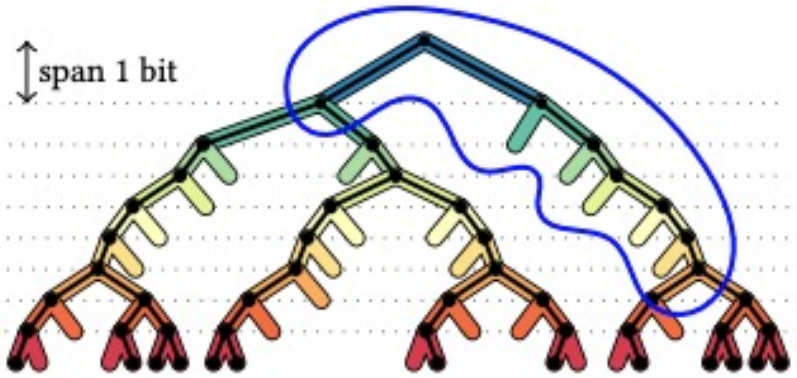
04 ／Height Optimized Trie（HOT）

02 Height Optimized Trie (HOT)

分布式存储与计算实验室

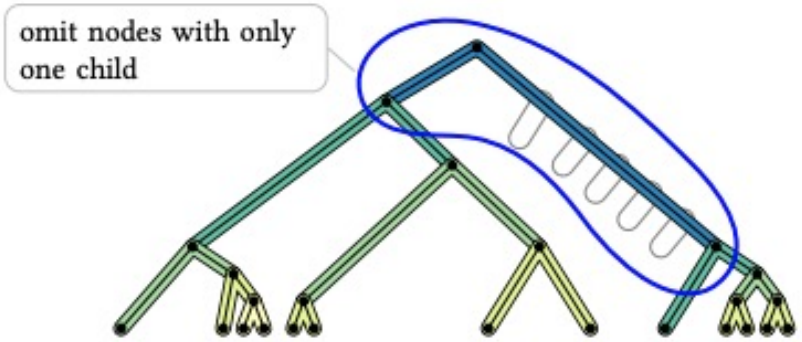
4.1 Height Optimized Trie (高度优化基数树)

HOT引入



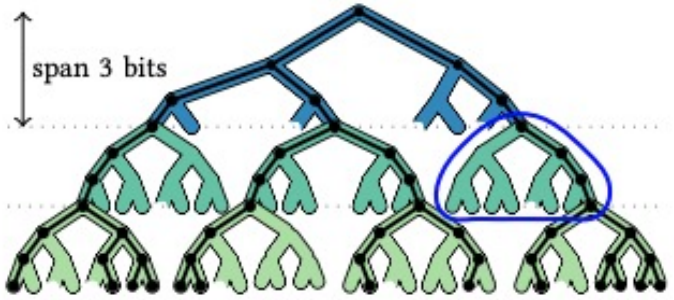
(a) Binary trie (height=9, #nodes=37).

压缩路径策略



(b) Patricia (height=5, #nodes=12).

增大span



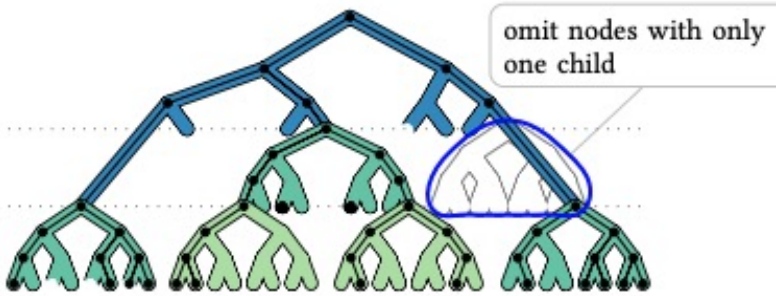
(c) 3 bit span (height=3, #nodes=8).

02 Height Optimized Trie（HOT）

分布式存储与计算实验室

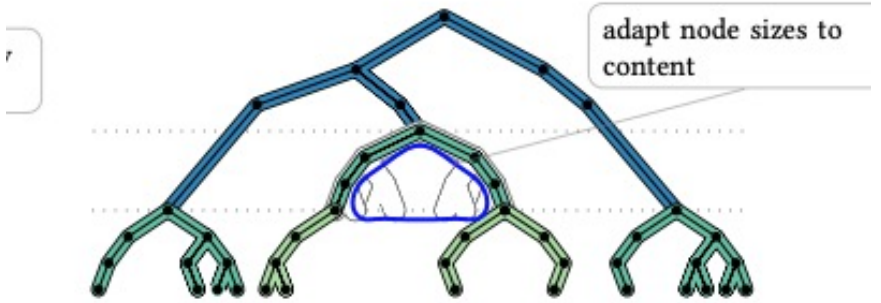
4.1 Height Optimized Trie（高度优化基数树）

HOT引入



(d) Patricia, 3 bit span (height=3, #nodes=6).

Adaptive Nodes



(e) Patricia, 3 bit span, adaptive nodes (height=3, #nodes=6).

问题

仍然存在多数空指针

现象

同层节点存储数据量差距较大

问题

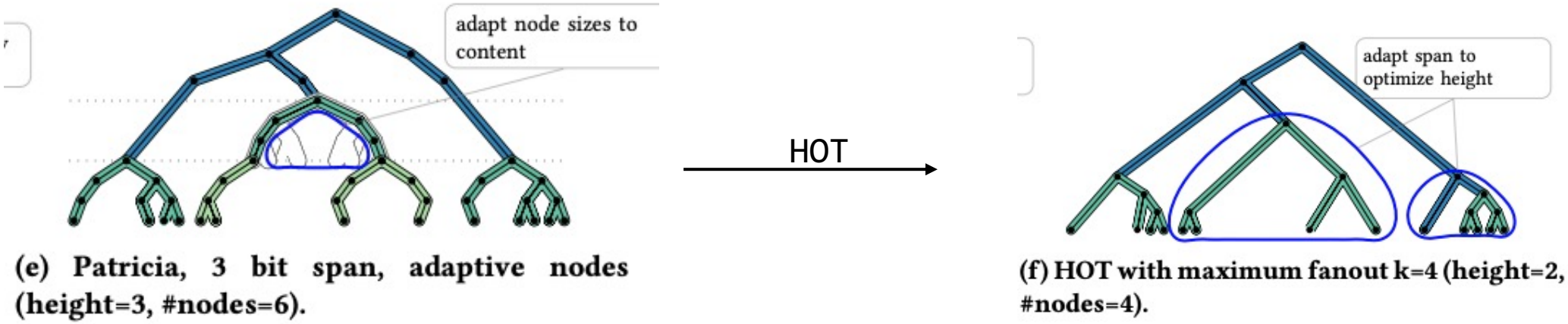
有些节点空间利用率较低

02 Height Optimized Trie（HOT）

分布式存储与计算实验室

4.1 Height Optimized Trie（高度优化基数树）

HOT引入



(e) Patricia, 3 bit span, adaptive nodes (height=3, #nodes=6).

(f) HOT with maximum fanout k=4 (height=2, #nodes=4).

fanout：每个节点包含的字符数

核心思想：自适应节点，动态span，固定fanout

优势：优化高度，减少节点空间浪费

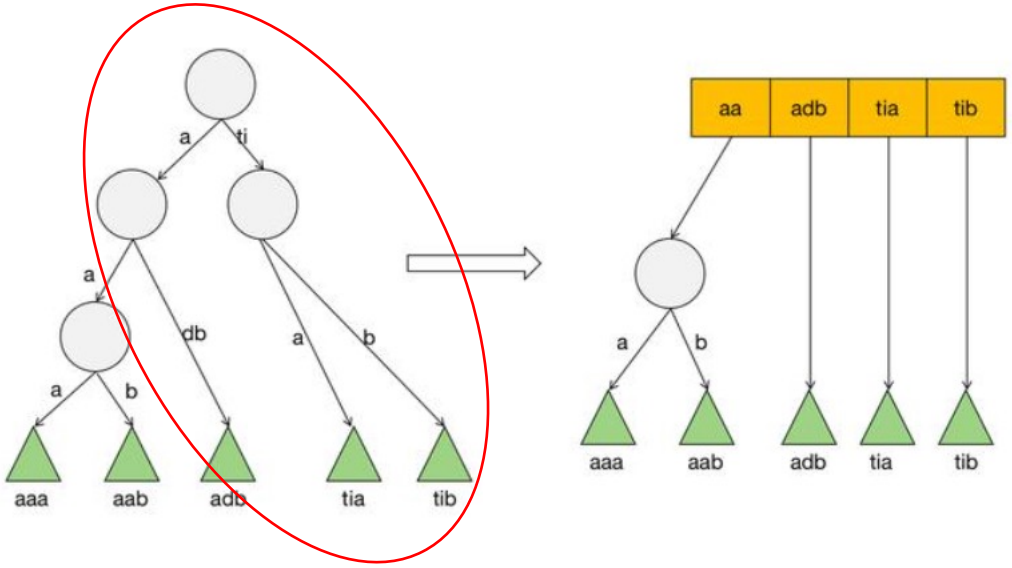
02 Height Optimized Trie（HOT）

分布式存储与计算实验室

4.1 Height Optimized Trie（高度优化基数树）

优化一：只存储特征点信息

1. 动态span导致字符宽度不一样
2. 如果不存储完整的aa，adb，tia，tib，
   怎样才能区分不同字符来找到下一层节点



将节点用二进制表示，只存储特征点信息

左分支为0，右分支为1

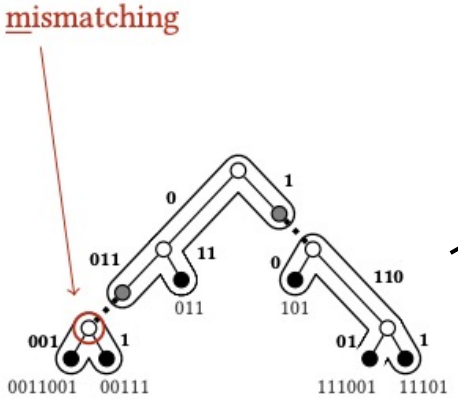
aa	0	1	1	0	0	0	0	1	0	1	1	0	0	0	0	1								
adb	0	1	1	0	0	0	0	1	0	1	1	0	0	1	0	0	0	1	1	0	0	0	1	0
tia	0	1	1	1	0	1	0	0	0	1	1	0	1	0	0	1	0	1	1	0	0	0	0	1
tib	0	1	1	1	0	1	0	0	0	1	1	0	1	0	0	1	0	1	1	0	0	0	1	0

02 Height Optimized Trie (HOT)

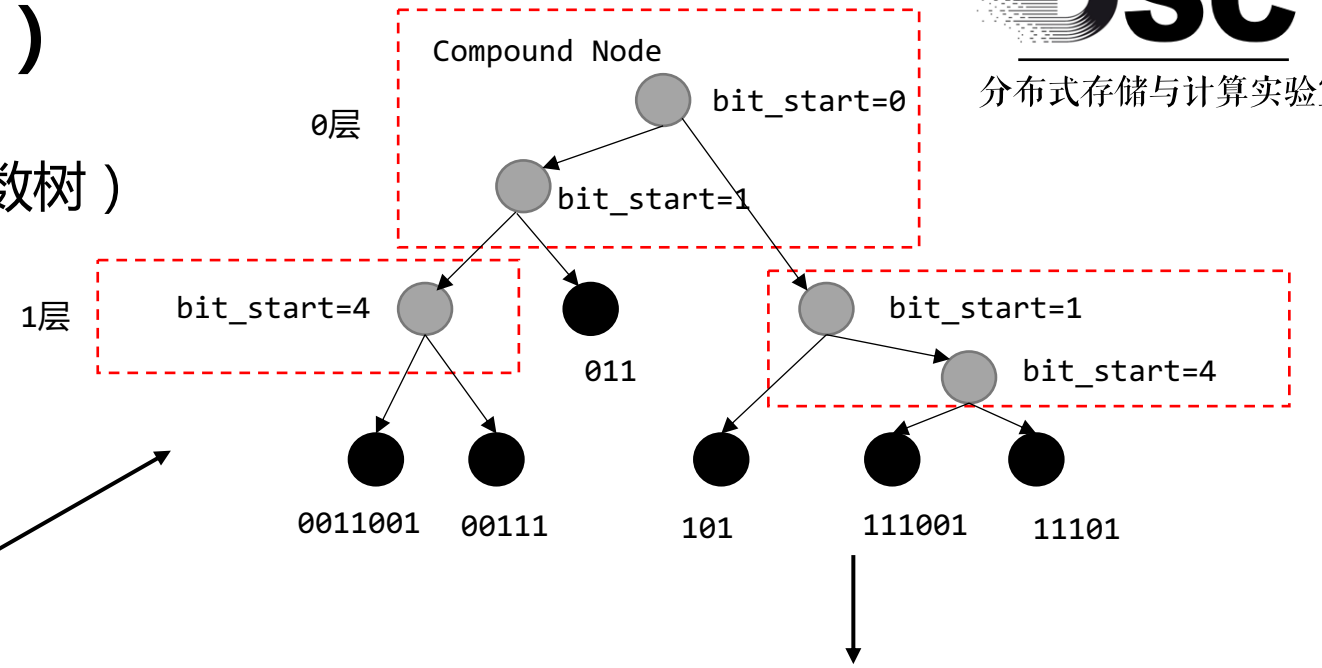
4.1 Height Optimized Trie (高度优化基数树)

HOT插入过程 —— Normal Insertion

核心思想：
1. 固定fanout
2. 尽量保持树高稳定

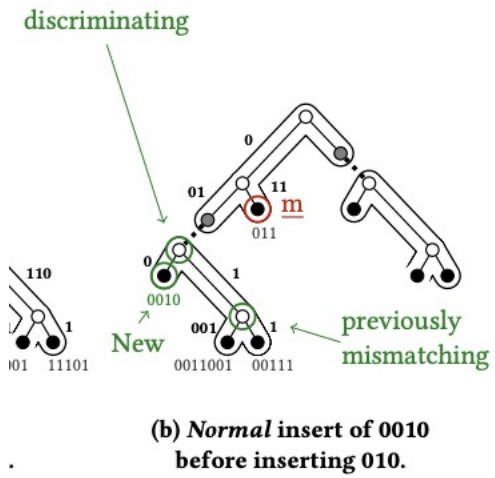


(a) Initial HOT before inserting 0010.



插入过程

1. 对于0010，依次找出区分点对比
2. 依次比较第0，1，4位的区分点
3. 0010不存在第4位，因此找到不匹配点
4. 插入节点，未影响树高



(b) *Normal* insert of 0010 before inserting 010.

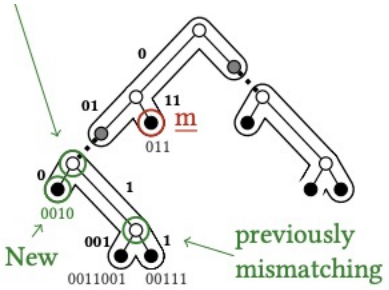
02 Height Optimized Trie（HOT）

分布式存储与计算实验室

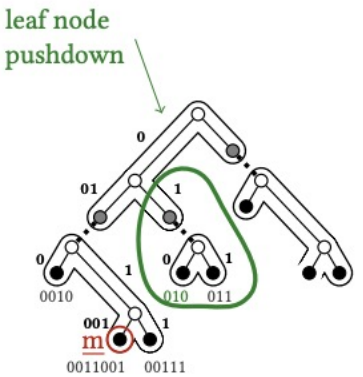
4.1 Height Optimized Trie（高度优化基数树）

HOT插入过程 —— Leaf Node Pushdown

1. 根据区分点找到不匹配节点是Leaf节点
2. 判断是否超过fanout，没超过，则Normal Insertion
3. 创建一个new Compound Node，并生成相应的BiNode
4. 插入节点，未影响树高



(b) *Normal* insert of 0010 before inserting 010.



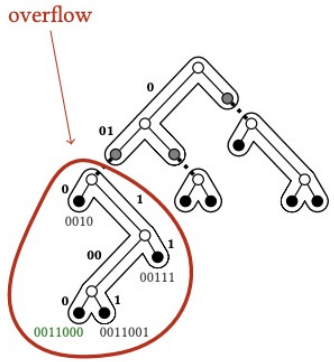
(c) Insert of 010 using *leaf node pushdown*. Before Inserting 0011000.

02 Height Optimized Trie (HOT)

分布式存储与计算实验室

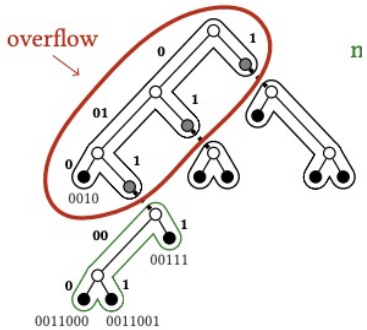
4.1 Height Optimized Trie (高度优化基数树)

HOT插入过程 —— OverFlow



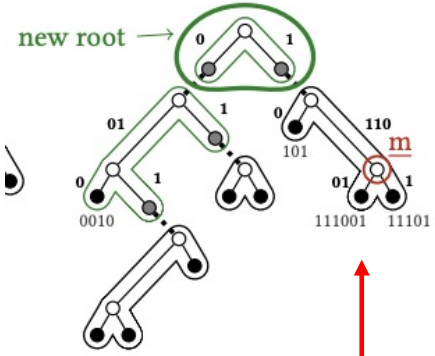
(d) Inserting 0011000 causes overflow in node n having $h(n) + 1 = h(parent(n))$.

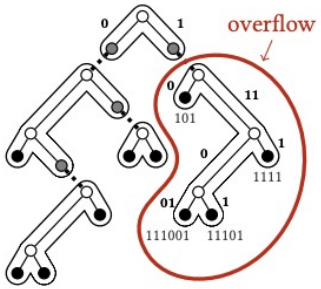
向上增长



(e) Handling overflow by *parent pull up* (propagates overflow to parent node).

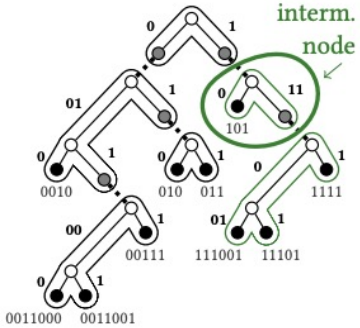
向上增长





(g) Inserting 1111 causes overflow in node n having $h(n) + 1 < h(parent(n))$.

自我分裂



(h) Overflow resolution by *intermediate node creation.*

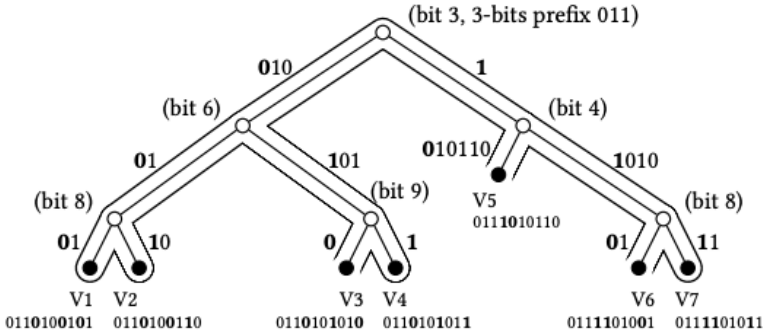
影响树高，则向上增长

02 Height Optimized Trie (HOT)

分布式存储与计算实验室

4.1 Height Optimized Trie (高度优化基数树)

HOT物理实现



(a) Example Binary Patricia Trie. Discriminative bits along a path are typeset in bold.

Raw Key	Bit Positions	Partial Key (dense)	Partial Key (sparse)
0 1 1 **0** 1 0 **0** 1 **0** 1	{3, 6, 8}	**0** 1 **0** **0** 1	**0** 0 **0** **0** 0
0 1 1 **0** 1 0 **0** 1 **1** 0	{3, 6, 8}	**0** 1 **0** **1** 0	**0** 0 **0** **1** 0
0 1 1 **0** 1 0 **1** 0 1 **0**	{3, 6, 9}	**0** 1 **1** 1 **0**	**0** 0 **1** 0 **0**
0 1 1 **0** 1 0 **1** 0 1 **1**	{3, 6, 9}	**0** 1 **1** 1 **1**	**0** 0 **1** 0 **1**
0 1 1 **1** **0** 1 0 1 1 0	{3, 4}	**1** **0** 0 1 0	**1** **0** 0 0 0
0 1 1 **1** **1** 0 1 0 **0** 1	{3, 4, 8}	**1** **1** 1 **0** 1	**1** **1** 0 **0** 0
0 1 1 **1** **1** 0 1 0 **1** 1	{3, 4, 8}	**1** **1** 1 **1** 1	**1** **1** 0 **1** 0

(b) Raw and partial keys for the trie in (a).

Partial Key(dense):基于所有特征点位置得出的具体特征点

Partial Key(sparse):基于具体路径上的特征点Bit Positions位置标识出来的特征点

实际使用Partial Key(sparse)，插入节点，代价小

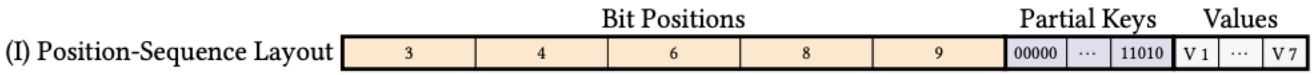
02 Height Optimized Trie（HOT）

分布式存储与计算实验室

4.1 Height Optimized Trie（高度优化基数树）

HOT物理布局

优化二：编码存储适应SIMD



不符合SIMD并行指令用法，在查找任意一个Key时，要先根据每一个Bit Position抽取Key对应的特征点，这是一个串行的过程，效率太低

为了SIMD操作，将Partial key对齐(8-bit, 16-bit和32-bit)

优势：并行处理，加快查询时间

05 / Hyperion

02

Hyperion

分布式存储与计算实验室

5.1 Hyperion（内存优化基数树）

目标

更优的内存占用

核心思想

顺序存储，紧凑的节点设计，适量分配空间，原地更新

缺点

I. 对写性能很不友好

02 Hyperion

分布式存储与计算实验室

5.1 Hyperion（内存优化基数树）

总体方案

1. 固定span为16
2. 设置容器
   I. 以容器为单位进行顺序存储
   II. 允许嵌套容器
   III. 容器以32B增长，防止过度分配

容器布局

1. Size：容器的大小
2. Free：剩余空间
3. Payload：存储的节点数据
4. J，S：跳表，分裂标志

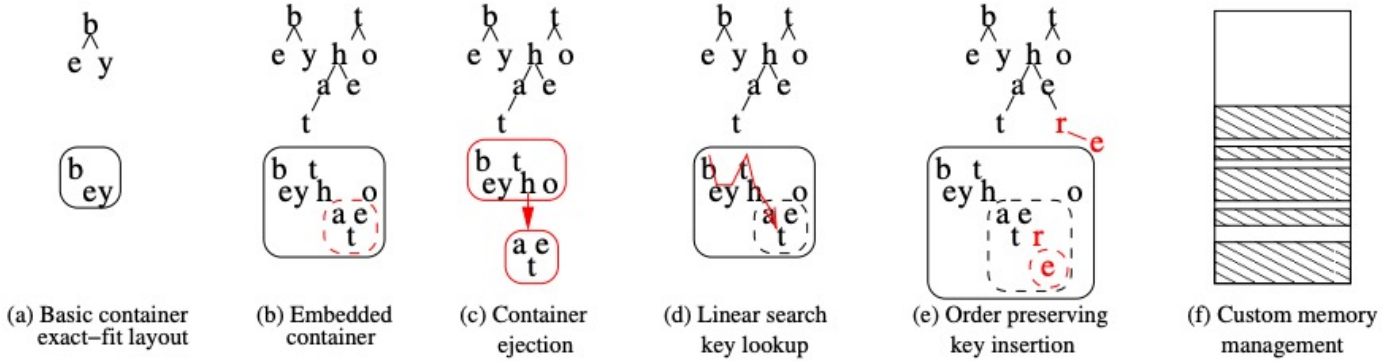


Fig. 2. Hyperion core concepts. The red arrow in (d) shows the linear lookup steps for the key *"that"*.

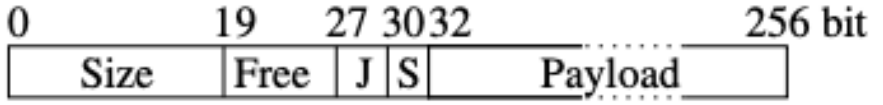


Fig. 3. Container: header and appended payload

02

Hyperion

分布式存储与计算实验室

5.1 Hyperion（内存优化基数树）

节点布局

1. 父节点(T-Node)与子节点(S-Node)分开存储，便于同级节点间跳转
2. T-Node与S-Node之间不存储指针
3. t：节点类型（内部节点 or 叶子节点）
4. k：T-Node or S-Node
5. c：判断是否存在下一个容器
6. d，js，jt：增量编码，跳转，跳表标志

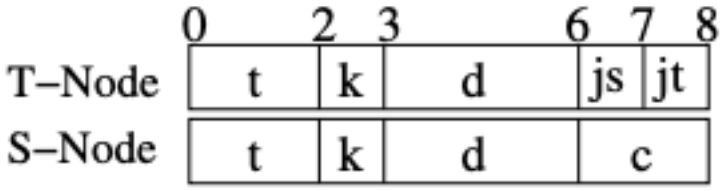
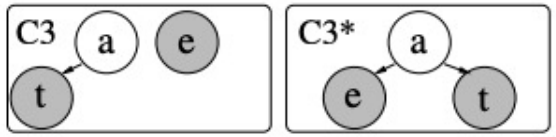
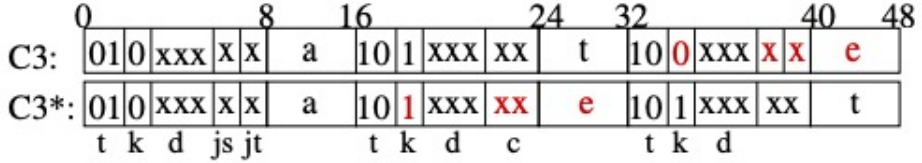


Fig. 5. Bit structure of the T/S-Nodes



(a) Graph representation



(b) Byte array representation

02 Hyperion

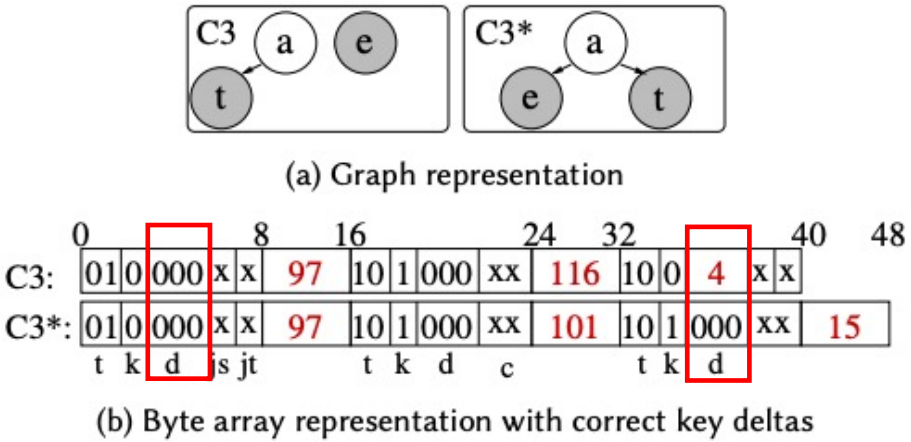
分布式存储与计算实验室

5.1 Hyperion（内存优化基数树）

优化一：Delta Encoding

1. 分别对T-Node，S-Node进行增量编码
2. 编码范围d：1～7
3. 编码成功，即无需存储具体值

优点：面向dense key场景，能大大降低内存消耗

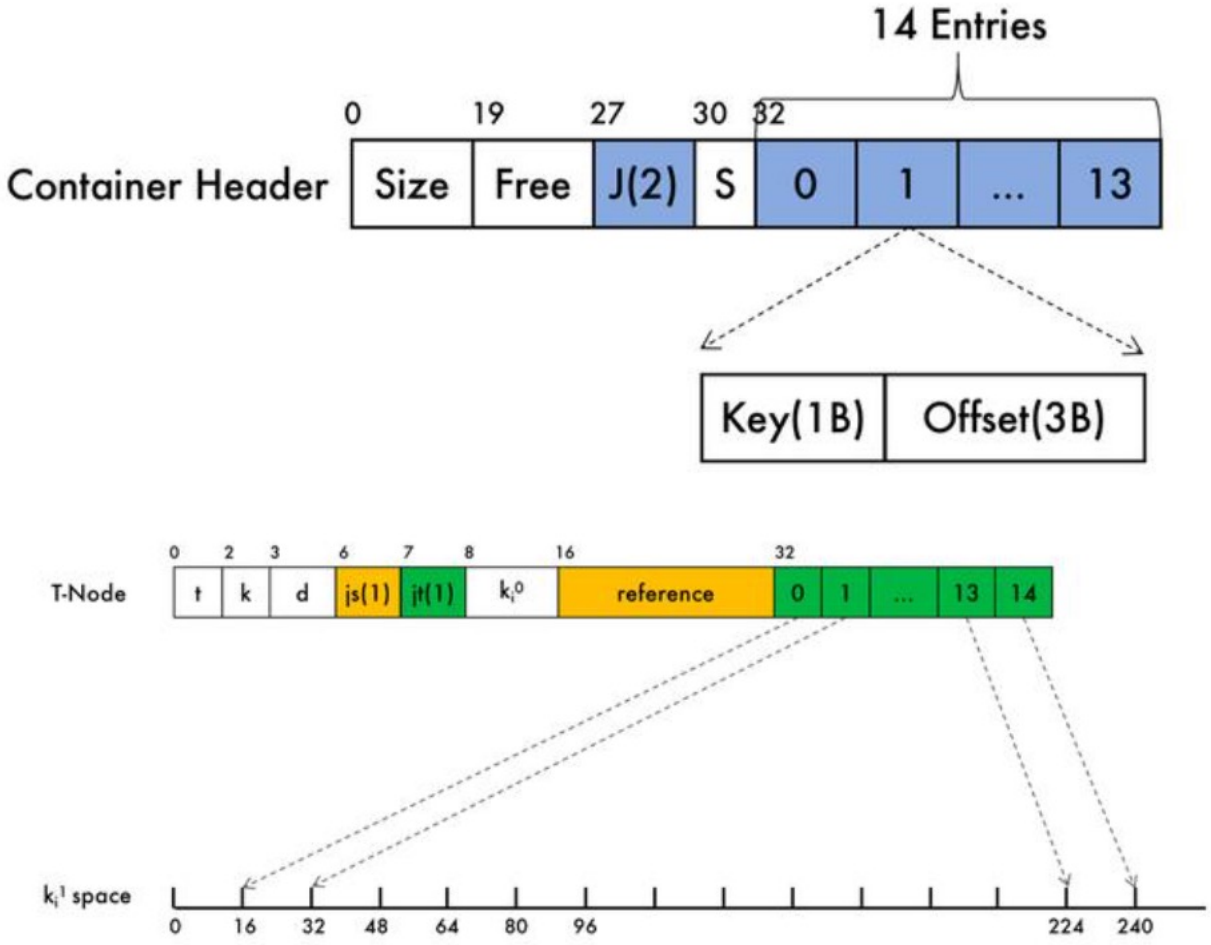


02 Hyperion

5.1 Hyperion（内存优化基数树）

优化二：查询优化（引入跳表）

1. Container Jump Table（jt）
   - I.为了快速定位T-Node
   - II.数组个数 = J*7
   - III.T-Node按数组个数等分
   - IV.定位到最接近目标的T-Node
2. T-Node Jump Table（jt）
   - I.为了快速定位S-Node
   - II.S-Node按数组个数等分
   - III.定位到最接近目标的S-Node



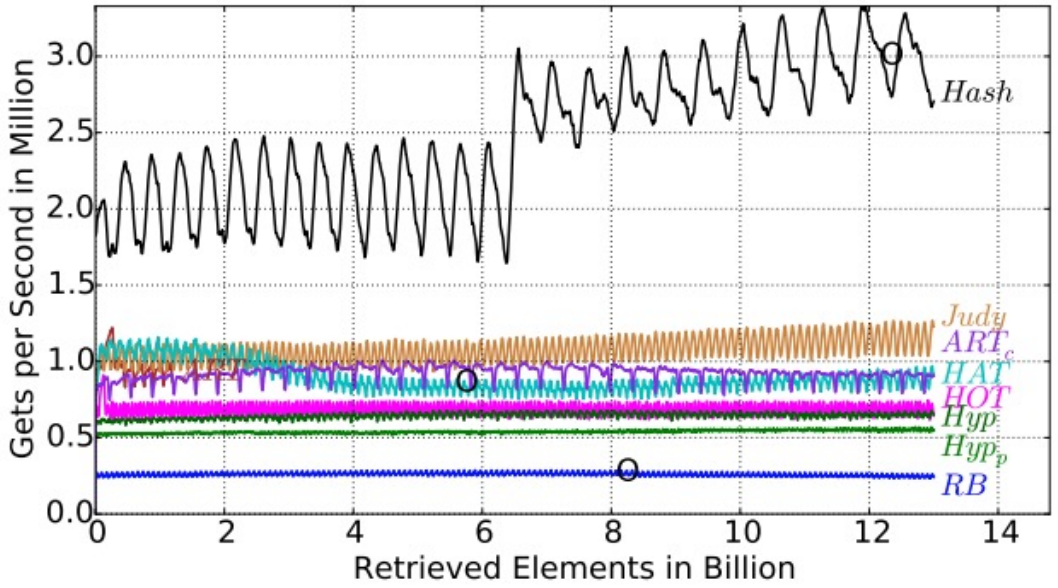
02

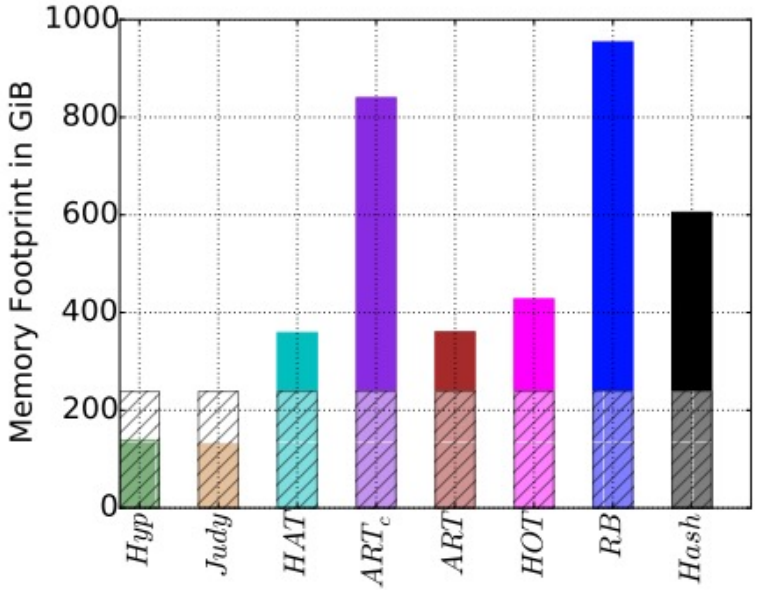
Hyperion

分布式存储与计算实验室

5.1 Hyperion（内存优化基数树）

性能对比





06 总结

05 总结

	思路	优点	缺点
Adaptive Radix Trie（ART）	1. 二进制表示字符 2. 增大span并固定 3. 自适应节点调整	1. 对于密集型数据，充分利用空间 2. 降低树高	1. 对于稀疏型数据，导致过多空间未利用 2. 树高可能严重不平衡
Height Optimized Trie（HOT）	1. 动态span 2. 固定fanout 3. 充分利用SIMD命令	1. 每个节点空间充分利用 2. 树高保持稳定	1. 实现复杂
Hyperion	1. 顺序存储 2. 小额分配 3. 跳表机制	1. 更优的内存占用	1. 插入，删除操作会引起大量数据移动

05 文献

DSC
分布式存储与计算实验室

[01] - Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13). IEEE Computer Society, USA, 38–49.

[02] - Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18). Association for Computing Machinery, New York, NY, USA, 521–534.

[03] - Markus Mäsker, Tim Süß, Lars Nagel, Lingfang Zeng, and André Brinkmann. 2019. Hyperion: Building the Largest In-memory Search Tree. In <i>Proceedings of the 2019 International Conference on Management of Data(<i>SIGMOD ‘19). Association for Computing Machinery, New York, NY, USA, 1207–1222.