软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2019,30(11):3364−3381 [doi: 10.13328/j.cnki.jos.005579]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

分布式数据库下基于剪枝的并行合并连接策略*

高锦涛, 李战怀, 杜洪涛, 刘文洁

(西北工业大学 计算机学院,陕西 西安 710129)

通讯作者: 高锦涛, E-mail: gaojintao@mail.nwpu.edu.cn

摘 要: 排序合并连接是数据库系统一种重要的连接实现方式,比哈希连接有更广泛的应用.分布式环境下,数据分片、分布存储,面对昂贵的网络代价,进行高效排序合并连接的挑战巨大.传统策略首先针对连接数据进行排序,然后基于排好序的数据执行合并连接.这两部分操作均基于原始数据进行操作,通常情况下,原始连接数据存在无用数据块,这些数据块无需连接,但会增加额外开销,包括网络开销.随着数据量的增多,出现无用数据块的概率增大,额外开销随之增多.传统策略没有预先处理这些无用数据块.针对这个问题,提出一种分布式环境下基于剪枝的并行排序合并连接策略(parallel sort-merge join based on prune,简称 Pr_PSMJ).其特点是,连接发生之前高效完成对连接对象无用数据块的剪枝处理,提高整体连接效率.基本思想是,根据连接对象对应的连接分区数据统计信息,构造一种双边邻接表(bilateral adjacency list,简称 BAL),用来对连接数据中无用数据块进行剪枝,并保证最终连接结果的正确性;剪枝完成后,利用 BAL 计算出各个最佳本地连接执行点,并指导分区数据的迁移,使数据移动量最小;在连接阶段,由于 BAL 保证本地连接执行节点的独立性,因此能够轻松并行执行整个连接过程,并在每个连接点本地利用多核环境完成局部并行排序合并连接;最后,将局部结果合并成最终结果.由于 Pr_PSMJ 中的高效剪枝策略是在连接执行之前完成的,因此几乎适合任何合并连接操作,并且对于其他连接策略也有借鉴作用.给出了基于 Pr_PSMJ 的算法的正确性、效率性以及适应性分析,并且给出实验验证,证明了在分布式大数据量排序合并连接情况下,Pr_PSMJ 相对于其他策略能够有效减少网络开销,并提高连接效率.

关键词: 分布式;排序合并连接;剪枝;双边邻接表;并行

中图法分类号: TP311

中文引用格式: 高锦涛,李战怀,杜洪涛,刘文洁.分布式数据库下基于剪枝的并行合并连接策略.软件学报,2019,30(11):3364−3381. http://www.jos.org.cn/1000-9825/5579.htm

英文引用格式: Gao JT, Li ZH, Du HT, Liu WJ. Strategy of parallel merge join based on prune in distributed database. Ruan Jian Xue Bao/Journal of Software, 2019,30(11):3364−3381 (in Chinese). http://www.jos.org.cn/1000-9825/5579.htm

Strategy of Parallel Merge Join Based on Prune in Distributed Database

GAO Jin-Tao, LI Zhan-Huai, DU Hong-Tao, LIU Wen-Jie

(School of Computer, Northwestern Polytechnical University, Xi'an 710129, China)

Abstract: Sort-merge join is an important implementation method of join in database system, and is more widely used than hash join. Under distributed environment, data is sharded and distributed across many nodes, and usually needed to be transmitted by network which is very expensive. Therefore, it is far more challenging to efficiently process sort-merge join in distributed database. Traditional strategy firstly sorts data, and then carries out merge-join based on sorted data, which are both related with original data. But original data usually has useless data blocks, which does not participate in join, but will increase the extra cost during join including network cost. The bigger

* 基金项目: 国家自然科学基金(61732014, 61672432, 61672434, 61472321); 陕西省基础研究计划(2017JM6104)

Foundation item: National Natural Science Foundation of China (61732014, 61672432, 61672434, 61472321); Natural Science Basic Research Plan of Shaanxi Province (2017JM6104)

收稿时间: 2017-07-27; 修改时间: 2017-09-29, 2018-02-28; 采用时间: 2018-04-04; jos 在线出版时间: 2018-04-16

CNKI 网络优先出版: 2018-04-16 11:00:01, http://kns.cnki.net/kcms/detail/11.2560.TP.20180416.1059.010.html

of data size, the higher of possibility of useless data blocks. Traditional sort-merge join strategy does not prehandle these useless data. In this study, a parallel sort-merge join is proposed based on prune, called Pr_PSMJ, which can efficiently prune the useless data ahead from join data, and improve the efficiency of join. Firstly, a bilateral adjacency list (BAL) is constructed by the statistic information from shards of join data. Using BAL, the useless data of join data can be pruned and the correctness of final join result is guaranteed. Secondly, after the pruning, the optimal local-join executing place can be computed by BAL, and the quantity of data mitigating among nodes is minimized. Finally, during the join step, for the independence of local-join guaranteed by BAL, the executing of sort-merge join can be easily parallelled, and in every executing node, it is natural to parallel the local-partitial-join using multi-core environment. The final result is achieved by merging local-result. Because high efficient prune operation is done before executing join, Pr_PSMJ is almost fit for every sort-merge strategy, and it is a good lesson for other join strategies. The correctness, efficiency, and adaptability of algorithm are analyzed based on Pr_PSMJ. By experiments, it is proved that under distributed environment, orienting large data, Pr_PSMJ can effectively decrease the overhead of network and improve the join efficiency than other strategies.

Key words: distribution; sort-merge join; prune; bilateral adjacency list; parallel

排序合并连接是数据库系统的一种重要的连接实现方式[1,2],比哈希连接有着更广泛的应用.分布式环境下,数据量巨大,数据分片、分布存储,导致连接过程中存在大量网络代价,因此,高效地进行大数据量排序合并连接,挑战巨大.根据经验及实验可得出,通常情况下,连接数据都可能存在无用数据块,即不需要进行连接的数据.随着数据量增大,无用数据块比例可能越来越高,增加额外开销,比如分布式环境下的网络开销,降低连接效率.

排序合并连接过程涉及取数据、排序、连接等步骤,集中式架构下执行这些步骤涉及 CPU 和 IO 代价,分布式环境下由于数据分片、跨域存储,需要额外考虑网络传输代价.以 OceanBase 数据库[3]为例,介绍分布式环境下集中式处理排序合并连接过程.OceanBase 中,连接数据分布在不同存储节点,连接之前,将分散的数据全部拉取到查询节点本地进行排序,排序完毕后进行合并连接,这种排序合并连接策略存在如下问题:(1) 没有对连接数据中无用数据块进行剪枝;(2) 在查询节点进行集中式排序;(3) 在查询节点进行集中式全局合并连接.在处理大数据量连接情况下,这些问题造成大量网络代价以及本地 CPU 和 IO 代价.一些文献[4−7]针对问题 2 和问题 3 提出了并行排序策略,将连接数据进行分区,分别迁移到多个进程上进行并行排序以及局部连接,最后全局合并连接的策略.但并没有针对第 1 个问题给出很好的解决策略.

排序合并连接需要连接数据有序,通过比较两边连接数据是否符合连接条件决定输出结果.在数据量大的情况下,两边连接数据大概率存在多个无效数据块,这些数据块不会产生输出结果,但会产生大量额外代价.如两个有序序列 A(−1000000,...−1,0,1,2,...,1000)和 B(−2000000,...,−1000001,0,1,2,...,1000)进行等值合并连接,按照传统策略,需要至少比较 1000000+1000×2 次.但 A 的子区间[−1000000,0]和 B 的子区间[−2000000,−1000001]无连接结果输出,为无用数据块,因此对于此区间内的比较完全没有必要,并且分布式环境下会增加额外昂贵的网络代价.如果能够将这些无用数据提前进行预处理,将其剪枝掉,将会大大减少连接代价.图 1 为在 OceanBase 中进行排序合并连接实验时未剪枝(normal)和人工剪枝(prune)前后性能对比,连接对象为两个数据量为 1 000 000 的字符串序列.其中,重复度指匹配连接的数据占原始数据的百分比.

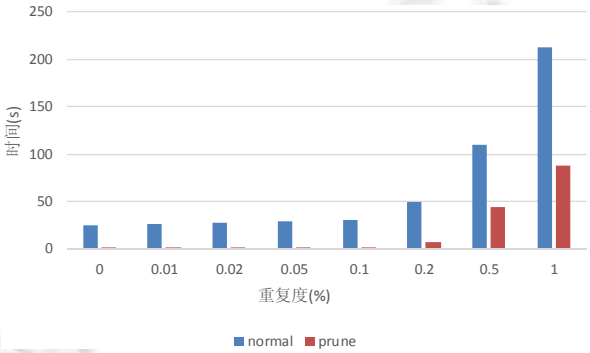

Fig.1 Performance comparation of merge-join between prune and non-prune

图 1 未剪枝与剪枝前后合并连接性能对比

可以看出,将无用数据剪枝后的连接性能远远优于剪枝前的连接性能.而面对复杂的数据特征,需要成熟的剪枝策略.目前,排序合并连接优化策略主要包括将取数据和排序过程由串行变为并行,或者连接阶段将搜索范围缩小等措施.这些优化手段基于参与连接的原始数据进行后续处理,并没有对原始数据进行预处理.本文提出一种分布式数据库下基于剪枝的并行排序合并连接策略(Pr_PSMJ),针对数据分布信息以及分区数据统计信息,构造一种双边邻接表(bilateral adjacency list,简称 BAL),用来对连接数据中无用数据块进行剪枝,并保证最终连接结果的正确性.面对分布式环境,为了避免剪枝阶段的数据迁移,剪枝不能以整体连接数据为单位,而是以连接数据涉及的分片为单位;剪枝完成后,利用 BAL 计算出各个最佳本地连接执行点,指导分区数据的迁移,使数据移动量最小;在连接阶段,通过 BAL 保证各个本地连接执行节点的独立性,可以轻松并行执行整个连接过程,并且连接点内部能够利用多核环境进行局部并行排序合并连接.在分布式大数据量合并连接情况下,Pr_PSMJ 策略能够有效减少网络开销,并提高连接效率.

本文的主要贡献如下.

1) 给出一种分布式环境下基于剪枝的并行排序合并连接策略(Pr_PSMJ),能够对连接数据中无用数据块提前剪枝,并以最小数据迁移量完成本地并行合并连接,提高整体连接效率.
2) 给出 Pr_PSMJ 内容,并给出剪枝功能、本地连接中心以及切分因子构造方式.
3) 给出基于 Pr_PSMJ 的合并连接算法,并与经典算法在时间和空间上进行对比,给出算法正确性、效率性以及适应性分析,并结合实例给出 Pr_PSMJ 算法工作过程.
4) 在淘宝开源分布式数据库 OceanBase 中实现 Pr_PSMJ 策略,并给出实验评估,验证 Pr_PSMJ 策略的高效性.

本文第 1 节介绍排序合并连接相关工作.第 2 节给出并行排序合并的通用框架以及相关定义.第 3 节给出 Pr_PSMJ 策略的内容,包括剪枝功能、本地连接中心以及切分因子构造.第 4 节给出基于 Pr_PSMJ 策略的合并连接算法以及其他两个算法.第 5 节给出算法正确性、效率性以及适应性的分析,并给出示例.第 6 节给出实验评估.第 7 节给出本文的结论和未来工作.

1 相关工作

目前,针对排序合并连接的研究主要分为非阻塞式合并排序连接、多核环境下排序合并连接、分布式环境下排序合并连接.下面给出具体阐述以及相关讨论.

- 非阻塞式合并排序连接

排序合并连接是传统关系型数据库,如 Oracle、Sql Server、DB2 等的一项重要连接实现方式,其在真正执行连接前,需要保证连接数据有序,但排序阻塞连接执行.一些策略[8−10]假设连接数据已经准备好,假设前提是连接数据和连接执行都在本地,但面对大数据量或者网络应用,连接数据可能需要耗费较多代价得到.为了提高连接效率,一些文献提出非阻塞式合并排序连接[11−15].文献[13]提出一种 PMJ(progressive merge join)策略,保证快速给出连接的前几条数据,其他数据异步排序.文献[16]提出一种 HMJ(hash-merge join)算法,分为两个步骤:哈希和合并,哈希阶段对已经获得的数据利用内存哈希连接算法快速得到连接结果,如果连接数据出现阻塞,利用合并连接产生结果.其他合并连接策略包括流水线技术[17]、并行非阻塞连接[18]等.虽然这些策略能够提高发生阻塞时的连接效率,但并没有对连接原始数据进行剪枝.

- 多核环境下排序合并连接

随着硬件的快速发展,多核机器越来越普遍.为了充分利用多核环境下并行的执行优点,一系列并行执行策略[4−7]被提出来.文献[4]提出使用多核(4 096 个核),利用 MPI 进行合并连接的实施环境,通过合理规划网络资源,将排序和连接分离提高连接效率.文献[5]提出一种 P-MPSM 算法,利用多核对连接数据并行排序,并对左侧连接数据分区,并利用直方图进行重分区,处理数据倾斜问题,利用插补搜索[19,20],缩小连接范围,但插补搜索假设搜索的对象数据分布均匀.

- 分布式环境下排序合并连接

大数据时代,分布式数据库是处理和管理海量数据的利器,如 Google 的 spanner[21]、淘宝的 OceanBase 数据

库[3]等.分布式环境下,数据分片、分布存储,排序合并连接在取数据、排序、连接各个阶段涉及的数据量可能都较大,并且存在昂贵的网络传输代价,对于进行高效排序合并连接提出更大挑战.淘宝的 OceanBase 数据库中,排序合并连接过程为:并行获取连接对象的分区数据,并将分区数据发送到查询节点.虽然采用流水线执行方式,但存在单点内存全量排序的缺点,导致对于大数据量表的等值连接,效率较差(图 1).文献[22]针对 OceanBase 读写分离引起的数据合并代价,提出一种基线与增量数据分离架构下的排序归并连接优化算法,通过数据迁移达到连接数据的本地排序和连接.

- 讨论

非阻塞式并行连接能够减少连接时等待时间,但分布式环境下,网络传输代价为主宰代价,因此这些策略本质上并没有太多提高排序、连接效率.虽然多核环境和分布式环境下能够利用并行策略提高排序合并连接效率,但没有对无用数据块进行预处理,造成额外代价.本文提出的 Pr_PSMJ 策略能够预先对无用数据进行剪枝,减少局部连接和全局连接代价,并能够以最小数据移动量完成本地并行合并连接,提高整体的执行效率.

2 预备知识

为了针对分布式环境阐述本文提出的 Pr_PSMJ 策略,总结出一种通用的分布式框架,如图 2 所示.

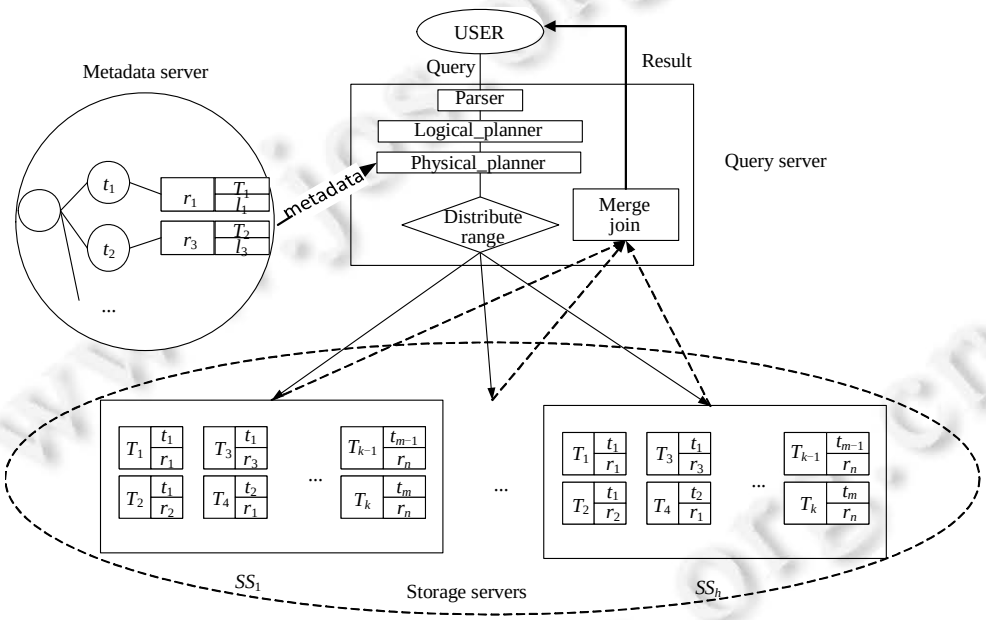

Fig.2 General distributed architecture

图 2 通用分布式框架

对各部分解释如下.

- Query server(QS):负责接收用户输入的 SQL 语句,并进行语法解析(parser)、逻辑计划生成(logical_planner)、物理计划生成(physical_planner)、范围分发(distribute range)、合并连接(merge join)等功能.合并连接的最终执行发生在 QS.
- Metadata server(MS):负责提供数据分片存储位置等元数据信息.
- Storage servers(SS):负责存储、操作(如查询)数据.存储模式为分布式,类似于 BigTable[23],每一个分布式节点存储一个表的部分或全部信息,并部署一个线程池$\{W_i\}$,动态地分配所需线程,完成对应任务.

以下内容针对连接语义(1)进行阐述,即 R 和 S 两个关系在连接属性 x 上进行等值连接.

$$\sigma_{\theta_1}(R)\infty_{R.x=S.x}\sigma_{\theta_2}(S) \tag{1}$$

定义 1(数据模式). 数据逻辑上以表为单位,物理上将表进行分片,分布存储在各个 *SS* 上.定义为公式(2).

$$\left.\begin{aligned} &DS=\{\{t_i(T_j:SS_k)\}\mid 0<i\leqslant n\wedge 0<j\leqslant m\wedge 0<k\leqslant s\}\\ &\bigcup_{j=1}^{m}T_j=t_i,T_g\cap T_h=f,0<g,h\leqslant m \end{aligned}\right\} \tag{2}$$

其中,*DS* 表示数据模式;t_i 表示表;T_j 表示表 t_i 的一个分区,通常,T_j 大小固定(Hbase 默认 64MB);SS_k 表示 T_j 的存储位置;T_j 所有数据的并集为 t_i;*f* 表示两个分区的交集,如果按照主键进行分区,则 *f* 为空集.

定义 2(排序合并连接[24]). 数据库的一种连接实现方式,适合自然连接和等值连接.针对公式(1)中的两个关系 *R*(*y*,*x*)和 *S*(*x*,*z*),其中,*x* 为两关系的连接属性,*y* 和 *z* 分别为其他属性,形式化定义排序合并连接见公式(3).

$$MJ=Me(\{So(R,x),So(S,x)\},\theta_{R.x=S.x}) \tag{3}$$

其中,*MJ* 表示排序合并连接;*Me* 表示合并操作,即将两个有序的序列按照连接条件合并成一个有序序列;*So* 表示对连接属性上的数据进行排序操作,其输出为有序数据,θ为连接条件.

定义 3(并行排序合并连接). 将定义 2 中的 *So* 操作以并行方式实现,为图 1 中的$\{W_i\}$分配排序任务以及执行排序任务的线程数.将 *Me* 操作改造为并行操作 *PMe*,即首先进行局部 *Me*,然后进行全局 *Me*.形式化定义为公式(4).

$$\left.\begin{aligned} &PSMJ=PMe\left(\left(\bigcup_{i=1}^{N}PSo(R_i,x,W_i,n_i),\bigcup_{j=1}^{M}PSo(S_j,x,W_j,n_j)\right),\theta_{R.x=S.x}\right)\\ &PMe:\\ &1.\ M_1:migrate(R_i\rightarrow s_{\{l_1,\ldots,l_M\}}),i\in(1,N)\\ &2.\ M_2:migrate(S_j\rightarrow s_{\{l_1,\ldots,l_N\}}),j\in(1,M)\\ &3.\ \text{If } l_k \textit{ is complete} \text{ then } Me(l_k),k\in(1,K) \end{aligned}\right\} \tag{4}$$

其中,*PSo* 表示将关系 *R* 和关系 *S* 相关的分区数据中连接属性 *x* 上的排序任务分别分配 n_i 和 n_j 个工作线程进行并行排序,其中,W_i 和 W_j 分别表示分布式节点上的线程池,*N* 和 *M* 表示 *R* 和 *S* 相关分区数据所在的分布式节点的个数,R_i 和 S_j 分别表示第 *i* 个节点和第 *j* 个节点上 *R* 和 *S* 的分区个数,且 $num(R_i)=n_i$,$num(S_j)=n_j$.执行 *PMe* 的步骤包括:

(1) M_1 表示将各个 R_i 包含的分片迁移到合适的 S_i 所在节点.

(2) M_2 表示将各个 S_j 包含的分片迁移到合适的 R_i 所在的节点.

(3) 如果迁移后的节点完备(l_k is complete),即可开始本地合并连接,完备的节点之间执行不阻塞.

问题定义. 公式(4)中,*PSo* 操作涉及 CPU、IO 代价,*PMe* 操作涉及 CPU、IO 以及网络代价.这两部分代价与参与 *PSo* 和 *PMe* 操作的数据量直接相关,并且决定 *PSMJ* 的效率.因此,本文需要解决的问题为:减少 *PSo* 和 *PMe* 操作中涉及到的不必要数据,最小化数据移动,提升 *PSMJ* 效率.形式化定义见公式(5).

$$prune\left(\sum_{i=1}^{N}R_i,\sum_{j=1}^{M}S_j\right),\ \min(M_1)\wedge\min(M_2) \tag{5}$$

其中,*prune* 功能将 *R* 和 *S* 中以块为单位进行无用数据块剪枝,剪枝后,最小化公式(4)中的 M_1 和 M_2.

3 基于剪枝的并行排序合并连接策略

分布式数据库中,数据分片、分布存储,在进行并行排序合并连接过程中,首先对分散的数据进行局部排序[5]或者全局排序[14],如果数据量巨大,将涉及大量网络开销.为充分利用分散数据局部排序得到的有序数据范围,进行无用数据块的剪枝处理,本文提出高效的基于剪枝的并行排序合并连接策略(Pr_PSMJ),目的是解决公式(5)中给出的问题.中心思想是,高效构造双边邻接表(BAL),实现对连接数据的剪枝处理(公式(5)中的 *prune* 功能),提前去除无用数据,并通过 BAL 实现本地并行合并连接过程中数据迁移量最小($\min(M_1)$和$\min(M_2)$).下面就 Pr_PSMJ 策略内容以及双边邻接表(BAL)进行详细阐述.

图 3 为基于 Pr_PSMJ 策略改造后的处理框架,标红的模块为新添加部分.改造点包括:(1) 添加剪枝(prune)功能模块,在范围分发之前,将多余的范围剪枝掉,即去除连接执行时多余的数据块;(2) 将图 2 中的 distribute range 改造为 distribute BAL,功能从分发数据范围改为分发双边邻接表(BAL);(3) 在每一个 *SS* 上构建一个本地连接中心(local_join_center,简称 LJC),作用是以最小代价完成一部分连接任务,LJC 的计算详见第 3.2 节.LJC 包括两个子模块:collector 模块,根据 prune 模块生成的双边邻接表收集需要在此 *SS* 完成的连接任务中包括的数据块;allocator 模块,为需要连接的数据块分配合适的资源以供并行执行.

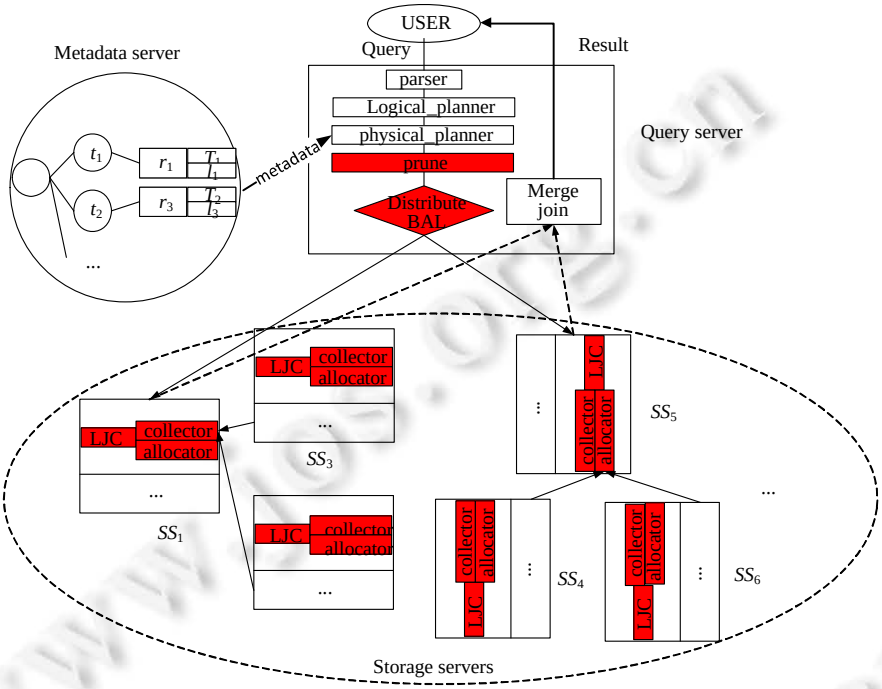

Fig.3 Modified distributed architecture based on Pr_PSMJ strategy

图 3 基于 Pr_PSMJ 策略改造后的分布式架构

定义 4. 基于定义 3 给出基于 Pr_PSMJ 的排序合并连接定义,见公式(6).

$$Pr\_PSMJ = PMe\left(Pr\left(\bigcup_{i=1}^{N} PSo(R_i,x,W_i,n_i),\bigcup_{j=1}^{M} PSo(S_j,x,W_j,n_j)\right),\theta_{R.x=S.x}\right) \tag{6}$$

添加剪枝(prune,简称 Pr)操作提取 *R* 和 *S* 对应分区数据的有序序列范围,建立 BAL,利用剪枝策略将无用数据块预先去除.并将 BAL 中各个项分发给对应 *SS*.进行本地合并后,由 *QS* 完成全局合并.

3.1 剪枝功能

如图 3 所示,剪枝功能(prune)作用为预先将无用数据块去除,功能包括双边邻接表(BAL)的构造(第 3.1.1 节)、边界处理(第 3.1.2 节)以及负载均衡(第 3.1.3 节).

3.1.1 BAL 构造

双边邻接表(BAL)的作用是完成剪枝功能,并利用 BAL 的结构特点,以每一个 BAL 项为单位,将整个分布式排序合并连接分割成独立的可并行执行的单元,结构如图 4 所示.BAL 分成 3 部分:左部、中部、右部.其中,左部关联左连接关系 *R* 相关的全部数据块范围集合$\{r_1,...,r_m\}$,中部为边界范围集合$\{ra_1,...,ra_n\}$,右部为剪枝后剩余的左连接关系 *R* 和右连接关系 *S* 对应的数据块范围.BAL 以中部各个元素作为候选本地连接中心(LJC),剪枝完成后,以中部对应的不为空的右部元素作为分发 BAL 的内容.$\{ra_1,...,ra_n\}$为边界范围集合,$\{r_1,...,r_m\}$为经过定义 4 中 *PSo* 操作后得到的关系 *R* 中有序数据块的范围集合.$\{s_1,...,s_n\}$为经过 *PSo* 操作后得到的关系 *S* 中的有序数

据块范围集合.{SS_1,...,SS_k}为图 2 中的存储节点集合.{$size_1$,...,$size_p$}为数据块范围集合对应的大小.BAL 左部表示关系 R 相关全部数据块的范围、该范围所在的位置以及对应数据量(根据直方图[12]或者样本估计[25]等策略得出的估计值),用三元组(r,l,$size$)表示.右部表示经过剪枝操作后剩余的集合{r}和集合{s}的部分,用四元组(s,SS,sub({r}),$size$)表示,其中 sub({r})表示与 s 完成本地连接所需的{r}的子集.注意,每一个{ra}元素并不一定有左部或者右部.

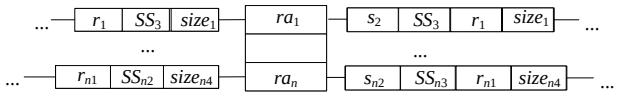

Fig.4 Architecture of BAL

图 4 BAL 架构

BAL 构造包括{ra}构造、左部构造和右部构造这 3 个部分,构造完成后的 BAL 架构如图 4.具体内容如下.

- 构造{ra}:首先获得 R 对应的数据块范围.做法为:在 R 相关的 SS 节点中进行 PSo 操作后得到有序数据块,同时获取其范围,形成集合{r},根据{r}得出{r}的超集 Sr,设置切分因子 q(设定方法详见第 3.3 节),将 Sr 切分成{ra}集合.经过以上处理,形成图 4 中{ra}.
- 构造左部:将已经获得的{r}映射到{ra}中,即将{r}中的每一个元素与{ra}元素取交集,如果交集不为空,则说明这个元素属于当前的 ra 元素.需要满足{r}中的每一个元素的全部或者部分(r 跨越{ra}某个元素的边界)只属于{ra}中的一个元素.经过以上处理,形成图 4 的左部,用{l_r}表示,其中每个元素为由(r_i,SS_j,$size_k$)组成的链表,每个{l_r}元素隶属于一个{ra}元素.
- 构造右部:使用和获取{r}同样的方法获取右连接关系对应的数据块范围{s},利用{s}中的每一个元素 s 探测当前形成的 BAL,形成两阶段探测:
  - 首次探测{ra},s 与{ra}的对应关系为 1 对多或者 1 对 1:如果为前者,则将 s 进行拆分,拆分后的子集映射到对应{ra}元素,这个过程中会将 s 的一部分无用数据舍弃;如果为后者,则直接映射到当前 ra 元素.
  - 再次探测 s 或者其子集在当前 ra 元素中对应的{l_r}中是否有对应的{r}与之相交:如果没有,则将这个 s 的全部或部分舍弃;如果有,则形成(s,SS,sub({r}),$size$)四元组;如果有多个,则形成链表.整个右部用{r_s}表示.

构造完成的 BAL 已经对无用的{r}和{s}数据块进行了剪枝,并且形成了以{r_s}中元素为单位的本地连接执行单元.

3.1.2 边界处理

由于{ra}是对{r}的超集进行切割形成的,因此在构造左部和右部时,可能存在{r}和{s}的元素跨越{ra}中多个元素,因此需要处理这种跨边界问题.处理策略为:以{ra}为基准,如果{r}或者{s}中的元素 r 或者 s 跨越了某个或多个{ra},则用{ra}的被跨越元素的边界值对 r 或者 s 进行切分,切分后,各个部分归属于其就近的较小{ra}.

3.1.3 负载均衡

负载均衡对于分布式数据库连接操作的并行执行效率至关重要,如文献[16,26,27]提出的并行环境下的负载均衡策略,能够有效处理某些情况,但并不适应分布式环境下的排序合并连接操作.Pr_PSMJ 策略通过 BAL 达到负载均衡目的.BAL 右部{r_s}中每一个元素对应一个连接执行点,多个元素之间并行执行,需要保证每个执行点的负载均衡.策略为选择合适的切分因子(详见第 3.3 节),并考虑 BAL 的每一个右部涉及到的数据量分布尽量均匀,达到各个执行节点的负载均衡以及降低本地连接时数据移动的网络代价.

3.2 本地连接中心

对连接关系完成剪枝操作后,需要根据形成的 BAL 的右部,执行图 3 中的 BAL 分发功能(distribute BAL),将右部各个元素发送到对应的存储节点(SS)上执行本地连接.为了保证连接的完备性,需要消耗网络代价将一

部分数据块迁移到执行本地连接的 *SS* 节点上,这个节点称为 LJC.被选择为 LJC 的 *SS* 节点通过 collector 模块,根据接收到的 BAL 右部元素对应的各个四元组,将不属于这个 LJC 的四元组对应的 *R* 和 *S* 的数据块收集到本地,然后利用 allocator 模块为收集完备的项分配对应的资源完成本地连接.为最小化迁移代价,提出公式(7)来计算 LJC.

$$LJC_k = \max_{i=1}^{x}\{size(sub_i(\{s\})) + size(sub_i(\{r\}))\} \tag{7}$$

公式(7)目的为选择本地参与连接的 *R* 和 *S* 数据块大小之和最大,其中,*x* 为第 *k* 个右部元素中所有四元组中涉及的执行节点的个数,$sub_i(\{s\})$和 $sub_i(\{r\})$分别表示第 *k* 个右部元素中关联的执行节点中关于关系 *R* 和 *S* 本地的的数据块个数.利用公式(7)依次计算出所有{*ra*}相关的执行节点,并由图 3 中 BAL 分发功能从 *QS* 分发到各个执行节点执行本地连接,发送过程和执行过程异步进行.

3.3 切分因子构造

切分因子作用为构造 BAL 中的{*ra*}部分进而限制{*r*}在 BAL 左部的分布({*l_r*})以及 BAL 右部{*r_s*}的形成,一定程度上决定了并行执行合并连接的节点数、负载均衡以及数据迁移量.图 5 阐述切分因子(*q*)在 BAL 关联的限制链(即链中前驱元素决定后继元素)中的位置,其作用范围为{*ra*}和{*l_r*}两个节点,但间接作用于其他后继节点.下面详细阐述切分因子的构造方法.

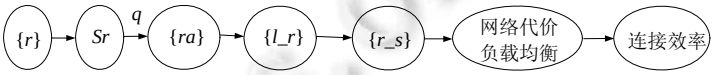

Fig.5 Restriction chain in BAL

图 5 BAL 限制链

构造切分因子所需参数包括{*r*}:{SS_r}:{*r_size*},{*s*}:{SS_s}:{*s_size*},其中,{SS_r},{SS_s}分别表示{*r*}和{*s*}对应的存储位置,{*r_size*}和{*s_size*}分别表示{*r*}和{*s*}中的每一个元素对应的数据量大小.这些参数在构造 *q* 之前已经具备.图 5 可以看出,*q* 的选择最终决定了{*r_s*}的构造,并且在使用 BAL 指导数据迁移时,根据公式(7),选择的是{*r_s*}中每一个元素所包含的所有{*r*}和{*s*}子集中在某一个执行节点里数据量最多的一个.由于构造{*r_s*}是在选择 *q* 之后进行的,因此为尽量最大化公式(7)并减少数据迁移,进行如下步骤选择 *q*.

- 提取{SS_r}和{SS_s}中每一个存储位置涉及的{*r*}和{*s*}子集以及对应的子集个数,形成{*loc*:*num*:*range*}.
- 根据{*r*}和{*s*}以及{*r_size*}和{*s_size*}计算出 *range* 对应的数据量之和,形成{*loc*:*num*:*sum*}.
- 计算 $avg_{num}=avg(num)$以及 $avg_{sum}=avg(sum)$.
- 由于针对分布式环境,因此以减少网络传输代价为首要目标,得出如下 *q* 计算公式.

$$q = \min\left(\sum_i \lceil num_i / avg_{num} \rceil, \sum_j \lceil sum_j / avg_{sum} \rceil\right) \tag{8}$$

从公式(8)可以看出,如果连接数据分布不均匀,则得出的 *q* 会相对较大,使{*ra*}粒度相对较小,能够将负载不均衡的节点关联的负载分摊到其他节点,较好地解决数据倾斜问题.

4 算 法

传统的排序合并连接优化策略基于连接对象关联的原始连接数据进行取数据、排序、连接等操作的优化,而忽略原始连接数据本身存在的无用数据块.在分布式环境下进行大数据量排序合并连接时,这些无用数据块将造成大量额外不必要代价.本文提出的基于剪枝的并行排序合并连接策略(Pr_PSMJ)通过构造双边邻接表(BAL),能够对原始连接数据进行高效剪枝,提前去除无用数据块,并通过公式(7)选择合适的 LJC,最小化数据迁移代价.为了显示 Pr_PSMJ 算法的优势,阐述 3 种算法进行对比,包括:uPr_uLJ(无剪枝无本地连接)算法,如 OceanBase[3]中的排序合并连接算法;uPr_LJ(无剪枝有本地连接),如文献[5]提出的 B-MPSM 算法以及本文提出的 Pr_LJC(有剪枝基于 LJC 的本地连接)算法.下面分别进行介绍.

uPr_uLJ 算法.

输入: $\sigma_{\theta_1}(R)\infty_{R.x=S.x}\sigma_{\theta_2}(S)$.

输出:合并结果.

1) (M_R,M_S)=*get_metadata*(R,S) //获取 R 和 S 对应元数据信息
2) fetch ($\{D_R\}$,$\{D_S\}$) by (M_R,M_S) parallel //根据元数据信息并行获取 R 和 S 对应的连接数据块
3) send ($\{D_R\}$,$\{D_S\}$) to *QS* //将取出的数据块全部发送到查询节点
4) *global_sort*($\{D_R\}$,$\{D_S\}$) in *QS* //在 *QS* 完成全局排序
5) *merge_result*=*global_merge*(*L_res*) //由 *QS* 将局部结果合并成全局结果
6) **return** *merge_result* //由 *QS* 返回用户最终执行结果

uPr_uLJ 算法首先获取连接关系 R 和 S 对应的元数据信息(第 1 行),然后根据元数据信息并行的获取到对应的数据块(第 2 行),并将获取到的数据块全部发送到查询节点(第 3 行)完成全局排序(第 4 行),最后在查询节点完成全局合并连接,将结果返回(第 5 行、第 6 行).

uPr_LJ 算法.

输入: $\sigma_{\theta_1}(R)\infty_{R.x=S.x}\sigma_{\theta_2}(S)$.

输出:合并结果.

1) (M_R,M_S)=*get_metadata*(R,S) //获取 R 和 S 对应元数据信息
2) fetch ($\{D_R\}$,$\{D_S\}$) by (M_R,M_S) parallel //根据元数据信息并行获取 R 和 S 对应的连接数据块
3) *local_sort*($\pi_x(\{D_R\},\pi_x\{D_S\})$) parallel //对数据块按照连接属性 x 并行排序
4) $\{W\}$=*allocate_work_Thread*(·) //为$\{DR\}$分配工作线程,线程数等于$\{DR\}$和$\{DS\}$中较大的元素个数
5) send ($\{S_R\}$) to $\{W\}$ //做$\{S_R\}$与$\{W\}$的一一映射,即每一个 W 接收一个 S_R
6) **For** every $W[i]$ **do** //对于每一个工作线程
7) *l_res*[*i*]=*local_merge*($W[i]$) //执行本地连接,以每一个 D_R 为中心,将需要与这个 D_R 连接的 S_R 从其他工作线程迁移到 D_R 所在线程,迁移完备以后,进行本地连接
8) **endFor**
9) *merge_result*=*global_merge*(*l_res*) //由 *QS* 将局部结果合并成全局结果
10) **return** *Merge_result* //由 *QS* 返回用户最终执行结果

uPr_LJ 算法的第 1 行、第 2 行与 uPr_uLJ 算法相同.为完成本地连接,在并行获取到数据块后,为关系 R 对应的每一个数据块分配一个工作线程(第 3 行),个数取决于较大的数据块个数,然后将关系 S 中的每一个数据块依次发送到工作线程中(第 4 行).对于每一个工作线程,为完成其本地连接,需要将此线程内 D_R 需要连接的所有 S_R 数据块迁移到本地完成本地连接(第 5 行~第 7 行).第 8 行、第 9 行与 uPr_uLJ 算法的第 5 行、第 6 行相同.

Pr_LJC 算法.

输入: $\sigma_{\theta_1}(R)\infty_{R.x=S.x}\sigma_{\theta_2}(S)$.

输出:合并结果.

1) (M_R,M_S)=*get_metadata*(R,S) //获取 R 和 S 对应元数据信息
2) fetch ($\{D_R\}$,$\{D_S\}$) by (M_R,M_S) parallel //根据元数据信息并行获取 R 和 S 对应的连接数据块
3) ($\{r\}$,$\{s\}$)=*local_sort*($\pi_x(\{D_R\},\pi_x\{D_S\})$) parallel //对数据块并行排序,获得连接属性对应的数据范围
4) send ($\{r\}$,$\{s\}$) to *QS* //将数据范围集合发送到查询节点(*QS*)
5) *BAL*=*generate_bal*($\{r\}$,$\{s\}$) //根据数据范围集合构造 BAL,完成剪枝操作
6) *L*=*get_merge_location*(*BAL*) //根据公式(7)获得最优本地执行点
7) distribute *BAL*[*i*].*right_part* to *L*[*i*] //将 BAL 的右部发送到本地执行点中
8) **for** every *L*[*i*] do //对于每一个执行点
9) *l_res*[*i*]=*local_merge_by_LJC*(*BAL*[*i*].*right_part*) //获得本地执行的结果

10) **endFor**

11) *merge_result=global_merge*(*L_res*) //由 *QS* 将局部结果合并成全局结果

12) **return** *merge_result* /由 *QS* 返回用户最终执行结果

Pr_LJC 算法的第 1 行、第 2 行与 uPr_LJ 算法的第 1 行、第 2 行相同.不同的是,需要通过本地并行排序的结果获取对应数据块的范围,并将它们发送到查询节点(第 3 行、第 4 行).在查询节点,根据第 3.1.1 节的内容生成 BAL(第 5 行),根据第 3.2 节的内容获得 BAL 右部对应的最佳本地执行点(LJC)(第 6 行).将 BAL 的右部分发到对应的 LJC(第 7 行),对于每一个 LJC,根据接收到的 BAL 右部内容将对应的数据块拉取到本地,完成本地连接任务(第 8 行~第 10 行).第 11 行、第 12 行与 uPr_LJ 算法的第 9 行、第 10 行相同.

5 算法分析

5.1 算法正确性

对于非剪枝、非局部连接算法(uPr_uLJ)以及非剪枝局部连接算法(uPr_LJ),其连接策略为:将连接的两表根据连接属性将参与连接的数据全部拉取到 QS 本地或者部分 SS 本地,进行全局或者局部排序,然后进行全局或者局部合并连接,最后由 QS 进行合并连接.两种算法涉及的数据来自原始连接数据,能够保证连接正确性.对于基于 Pr_PSMJ 策略的 Pr_LJC 算法,为了保证连接效率,通过剪枝去除连接数据中无用数据块,通过基于 LJC 的本地并行连接提高连接效率.在这个过程中,算法通过如下的细节保证连接的正确性.

- 保证对原始连接数据进行剪枝不影响最终结果的正确性.

两表原始连接数据分片范围分别为{*r*}和{*s*},通过各自范围内数据的并集能够得到原始连接数据.根据构造的 BAL(构造过程见第 3.1.1 节)对{*r*}和{*s*}进行剪枝操作,剪枝过程为:首先将{*r*}合并后得到的范围根据切分因子(见第 3.3 节)进行划分,得到{*ra*};然后将{*r*}中的元素映射到对应{*ra*}内,用{*s*}中的某个元素 *s* 首先对{*ra*}进行匹配,如果匹配到,则进一步匹配其中的{*r*}元素,如果也匹配到,则保留 *s*.如果这两次匹配中任何一个不成立,则舍弃 *s*.当某一个 BAL 右部不再增长时,说明此右部对应的范围内的数据已经完备,剩余的{*r*}和{*s*}为该范围内进行局部连接的数据.基于公式(9),其中 *N* 为{*ra*}内元素个数,可以得到在每一个{*ra*}元素内进行的利用{*s*}中的元素对{*ra*}元素内的{*r*}元素进行剪枝是互相独立的;同理,对于每一个{*ra*}元素内的剪枝活动,去除的{*r*}元素和{*s*}元素相对于其他{*ra*}元素也是独立的,并且每一个{*ra*}内的{*r*}元素是完全的且独立的,剪枝掉的数据块确实为无用数据块,因此整个剪枝活动对于排序连接的正确性没有影响.

$$\left.\begin{aligned}&\text{based on}: \bigcap_{0<i,j<N,i\neq j}(ra_i,ra_j)=\varnothing\\&infer: \bigcap_{0<i,j<N,i\neq j}(\{r\}_i,\{r\}_j)=\varnothing\\&infer: \bigcap_{0<i,j<N,i\neq j}(\{s\}\xrightarrow{map}\{r\}_i,\{s\}\xrightarrow{map}\{r\}_j)=\varnothing\end{aligned}\right\} \tag{9}$$

- 保证剪枝后的本地并行连接的正确性.

由于每个非空 BAL 的连接是独立且完备的,并且对应的连接数据完全来自基于公式(9)已经证明正确的数据基础上进行的,并且最终由 *QS* 完成全局连接,因此能够保证在实行本地并行连接后最终连接结果的正确性.

5.2 算法效率性

针对 3 种算法进行计算和存储方面的效率评估.uPr_uLJ 算法涉及的连接步骤包括取数据(fetch data,简称 fd)、全局排序(global sort,简称 gs)以及全局合并连接(global merge join,简称 gmj),uPr_LJ 算法涉及的连接步骤包括取数据(fd)、局部并行排序(local parallel sort,简称 lps)、本地连接(local merge join,简称 lmj)、全局结果合并(global result merge,简称 grm),Pr_LJC 算法涉及的连接步骤包括取数据(fd)、局部并行排序(lps)、剪枝(prune,简称 pr)、基于 LJC 的本地连接(local merge join based on LJC,简称 lmjLJC)、全局结果合并(grm).3 种算法涉及的执行场地包括 *QS* 和 *SS*,涉及的资源包括 *QS* 本地 CPU(*QS_CPU*)、内存(*QS_Mem*)、IO(*QS_IO*)、*SS* 本地 CPU(*SS_CPU*)、内存(*SS_Mem*)、IO(*SS_IO*)以及 *QS* 和 *SS* 之间的一次网络通信代价(*C_Net*).下面分别给出 3 种算法

对应的代价计算公式.

- uPr_uLJ 算法相关的计算代价和存储代价的计算见公式(10).

$$\begin{array}{ll} \text{计算代价} & \text{存储代价} \\ C(uPr\_uLJ)=C\_fd_1+C\_gs_1+C\_gmj_1 & S(uPr\_uLJ)=S\_fd_1+S\_gs_1+S\_gmj_1 \\ C\_fd_1=\sum_{i=1}^{N}(SS_i\_IO+C_i\_Net+QS\_IO) & S\_fd_1=\sum_{i=1}^{N}(SS_i\_Mem)+QS\_Mem \\ C\_gs_1=QS\_CPU & S\_gs_1=QS\_Mem \\ C\_gmj_1=QS\_CPU & S\_gmj_1=QS\_Mem \end{array} \tag{10}$$

- uPr_LJ 算法相关的计算代价和存储代价的计算见公式(11).

$$\begin{array}{ll} \text{计算代价} & \text{存储代价} \\ C(uPr\_LJ)=C\_fd_2+C\_lps_2+C\_lmj_2+C\_grm_2 & S(uPr\_LJ)=C\_fd_2+C\_lps_2+C\_lmj_2+C\_grm_2 \\ C\_fd_2=\sum_{i=1}^{N}(SS_i\_CPU+SS_i\_IO) & S\_fd_2=\sum_{i=1}^{N}SS_i\_Mem \\ C\_lps_2=\sum_{i=1}^{N}SS_i\_CPU & S\_lps_2=\sum_{i=1}^{N}SS_i\_Mem \\ C\_lmj_2=\sum_{i=1}^{N}(SS_i\_CPU)+\sum_{j=1}^{M}(C_j\_Net) & S\_lmj_2=\sum_{i=1}^{N}SS_i\_Mem \\ C\_grm_2=\sum_{i=1}^{N}(C_i\_Net)+QS\_CPU & S\_grm_2=QS\_Mem \end{array} \tag{11}$$

- Pr_LJC 算法相关的计算代价和存储代价的计算见公式(12).

$$\begin{array}{l} \text{计算代价} \\ C(Pr\_LJC)=C\_fd_3+C\_lps_3+C\_pr_3+C\_lmjLJC_3+C\_grm_3 \\ C\_fd_3=\sum_{i=1}^{N}(SS_i\_CPU+SS_i\_IO) \\ C\_lps_3=\sum_{i=1}^{N}SS_i\_CPU \\ C\_pr_3=QS\_CPU+\sum_{i=1}^{K}C_i\_Net \\ C\_lmjLJC_3=\sum_{i=1}^{\mu\times N}SS_i\_CPU+\sum_{i=1}^{L}C_i\_Net,0<\mu<1 \\ C\_grm_3=QS\_CPU+\sum_{i=1}^{\mu\times N}C_i\_Net \\ \text{存储代价} \\ S(Pr\_LJC)=S\_fd_3+S\_lps_3+S\_pr_3+S\_lmjLJC_3+S\_grm_3 \\ S\_fd_3=\sum_{i=1}^{N}SS_i\_Mem \\ S\_lps_3=\sum_{i=1}^{N}SS_i\_Mem \\ S\_pr_3=QS\_Mem \\ S\_lmjLJC_3=\sum_{i=1}^{\mu\times N}SS_i\_Mem \\ S\_grm_3=QS\_CPU+\sum_{i=1}^{\mu\times N}C_i\_Net \end{array} \tag{12}$$

分析. 从公式(10)可以看出,uPr_uLJ 算法没有经过局部排序以及局部连接提升连接效率以及降低 *QS* 的压力,在数据量较大情况下,*QS* 会成为系统瓶颈.uPr_LJ 算法采用了局部排序以及本地连接的策略,从公式(11)可以看出,取数据以及排序所耗费的计算代价和存储代价较 uPr_uLJ 算法都有所减少,并且由于采用并行本地连接策略,因此在提升连接效率的同时,减少了在 *QS* 合并的数据量,降低了 *QS* 的计算和存储的压力.但 uPr_LJ 算法在执行本地连接之前并没有将无用的数据块去除,导致增加额外的计算和存储代价.本文提出的 Pr_LJC 算法增加了高效剪枝功能,提前将无用数据块去除,并且采用基于 LJC 的本地连接策略,最小化数据迁移带来的网络代价,使性能整体上优于 uPr_uLJ 算法和 uPr_LJ 算法.3 种算法的比较过程见公式(13)、公式(14).

$$\begin{aligned}&\text{based on}: C\_fd_2 < C\_fd_1 \wedge S\_fd_2 < S\_fd_1 \wedge\\ &\quad C\_lps_2 < C\_gs_1 \wedge S\_lps_2 < S\_gs_1 \wedge\\ &\quad (C\_lmj_2 + C\_grm_2) \leqslant C\_gmj_1 \wedge\\ &\quad (S\_lmj_2 + S\_grm_2) \leqslant S\_gmj_1\\ &\textit{infer}: \quad C(uPr\_LJ) < C(uPr\_uLJ) \wedge S(uPr\_LJ) < S(uPr\_uLJ)\end{aligned} \tag{13}$$

$$\begin{aligned}&\text{based on}: C\_fd_3 = C\_fd_2 \wedge S\_fd_3 = S\_fd_2 \wedge\\ &\quad C\_lps_3 = C\_lps_2 \wedge S\_lps_3 = S\_lps_2 \wedge\\ &\quad C\_lmjLJC_3 < C\_lmj_2 \wedge S\_lmjLJC_3 < S\_lmj_2\\ &\quad C\_grm_3 < C\_grm_2 \wedge S\_grm_3 < S\_grm_2\\ &\quad C\_pr \text{ and } S\_pr \text{ very small}\\ &\textit{infer}: \quad C(Pr\_LJC) < C(uPr\_LJ) \wedge S(Pr\_LJC) < S(uPr\_LJ)\end{aligned} \tag{14}$$

5.3 算法适应性

对于任何排序合并算法,都需要经历取数据、排序、合并的过程.在大数据时代,连接数据通常数据量巨大,并且分片存储(无论是集中式还是分布式).无论采取什么连接策略,处理对象基本都是排序后的原始连接数据,因此通过 Pr_PSMJ 对排序后的数据进行剪枝,势必会提高后续的合并效率,进而提高整体的连接效率,这在分布式数据库中尤为明显.由于剪枝策略主要涉及很少数据量的网络传输代价,与其提升的性能相比可忽略不计,即使通过BAL没有剪枝掉任何数据块或者剪枝掉少量数据块,但BAL最小化数据迁移量能够节省网络开销,提升整体连接效率,因此适应各种不同的排序合并连接策略.

5.4 基于Pr_PSMJ连接过程举例

为了便于理解基于 Pr_PSMJ 策略的排序合并连接过程,以如下过程进行讲解.为叙述方便,将定义 1 中的数据模式简化,规定 t_i 包括两表{*R*,*S*},两表在各自属性 *x* 上进行等值排序合并连接.经过并行本地排序后,得到 *R* 和 *S* 在 *x* 上的{*r*}和{*s*},见表 1.

Table 1 Instance of {*r*} and {*s*}

表 1 {*r*}和{*s*}实例

类别	取值			
{*r*}	[1,100]	[70,230]	[200,260]	[2100,2200]
	[2150,2550]	[2500,2600]	[4100,4200]	[4150,4550]
	[4500,4600]	[6100,6500]	[7400,8000]	[7800,8000]
{*s*}	[−1000,0]	[30,50]	[60,120]	[80,120]
	[270,800]	[2160,2600]	[2700,2900]	[4250,4700]
	[8100,8500]	[8420,9500]	–	–

从{*r*}和{*s*}对应的存储位置{SS_r}和{SS_s}中提取{*loc*:*num*},对应的存储位置见表 2.其中,k 和 w 分别是千行和万行的单位,并且假设每一行大小基本相等.

Table 2 Storage location of {*r*} and {*s*}

表 2 {*r*}和{*s*}存储位置

类别	取值			
SS_1	r_1(1k) s_2(1k)	r_5(1k) s_3(1k)	r_6(3k) s_7(1k)	r_{10}(5k) s_8(4k)
SS_2	r_3(2k) s_4(2k)	r_7(1w) s_6(1k)	r_{11}(3k) –	r_8(3k) –
SS_3	r_9(2w) s_{10}(1k)	r_4(1k) s_5(2k)	r_2(1k) s_1(1k)	r_{12}(1w) s_9(8k)

- 切分因子 *q* 的构造:根据表 2 可得出,LJC 分别为(SS_1:8:17k),(SS_2:6:21k),(SS_3:8:44k);依据第 3.3 节可得出 *q*=4.
- {*ra*}构造:根据第 3.1.1 节以及切分因子 *q* 构造{*ra*},首先,针对{*r*}构造其超集 *Sr*=[1,8000];然后,根据切分因子对 *Sr* 进行切分,得到{*ra*}={[1,2000),[2000,4000),[4000,6000),[6000,8000]}.
- BAL 左部构造:根据第 3.1.1 节的左部构造策略,利用{*r*}对{*ra*}元素进行探测,得到 BAL 左部,如图 6 所示.

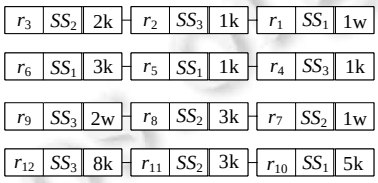

Fig.6 Instance of left part of BAL

图 6 BAL 左部实例

- BAL 右部构造:根据第 3.1.1 节 BAL 构造策略进行右部的构造.首先,利用获得的右连接关系 *R* 的数据范围集合{*s*},探测{*ra*},将{*s*}中的元素映射到{*ra*}集合中;然后,以{*ra*}中的元素为单位,在将对应的{*s*}元素与 *ra* 元素对应的{*r*}元素进行比较,得到 BAL 右部,如图 7 所示.

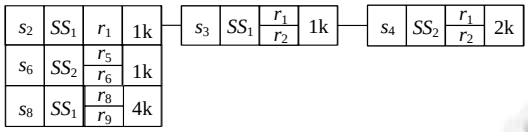

Fig.7 Instance of right part of BAL

图 7 BAL 右部实例

经过 BAL 的过滤,可将{r_3,r_4,r_7,r_{10},r_{11},r_{12}}和{s_1,s_5,s_7,s_9,s_{10}}无用数据过滤掉,节省大量网络代价以及 *SS* 和 *QS* 的本地 IO、CPU 代价和内存空间.再根据 BAL 进行局部合并阶段,依据公式(7)确定每一个非空右部对应的{*ra*}中的每一行所在的执行节点位置,可得{ra_1,l_1},{ra_2,l_2},{ra_3,l_1},根据计算的位置进行数据迁移,保证本地连接的完备性.每个节点在迁移完毕后,由本地节点的线程池分配等于{*ra*}元素对应的右部中链表个数的线程数以供局部并行连接,线程数分别为(2,1,1).局部连接完成后,将本地合并结果发送到 *QS*,完成全局合并,将合并结果返回给客户端.

6 实验评估

6.1 实验环境

本文实验使用 8 个计算节点,每个节点配置为:主频为 1400MHz 的 AMD Opteron(TM)处理器和 16GB 内存,

物理 CPU 个数为 2,物理核数 8,逻辑核数 16;操作系统为 Red Hat 6.2.所有算法均由 C++实现,算法实现平台为淘宝的开源分布式数据库 OceanBase 0.4 版本[28],系统架构如图 8 所示,其中,RootServer 提供元数据服务,主要包括数据分布信息;UpdateServer 提供 OceanBase 唯一的更新入口;MergeServer 提供对 SQL 语句的解析、逻辑计划生成、物理计划生成、计划分发、结果合并等功能;ChunkServer 提供数据的存储和查询等服务.OceanBase 具体描述请参见文献[3].图 2 与 OceanBase 模块的对应关系为:MetadataServer 对应 RootServer,QueryServer 对应 MergeServer,StorageServer 对应 ChunkServer.

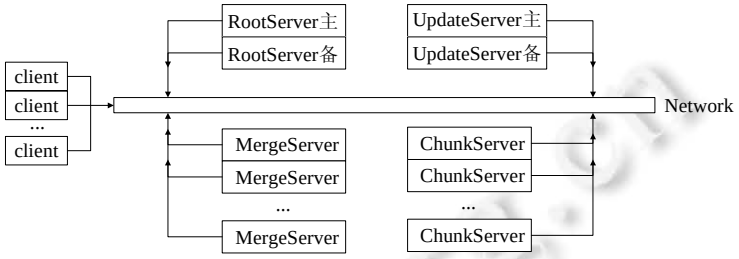

Fig.8 Architecture of OceanBase

图 8 OceanBase 架构

6.2 实验数据

本文实验数据由 tpch_2_17_0 工具生成,SF 设置为 30,选取 PART 表作为实验数据来源,PART 表数据量为 600 万行,大小为 699 兆.在 OceanBase 数据库中建立两张表作为连接测试表,分别为 *l_part* 和 *r_part*,并通过设置 *TABLET*_MAX_*SIZE* 参数指定表对应分块大小为 20MB.这两张表的结构和 tpch 生成的 PART 表结构一致.利用 OceanBase 的 import 工具将生成的数据分别导入到 *l_part* 和 *r_part* 中.

6.3 评估与结果分析

本文设计了 4 组实验,分别为:(1) 测试在重复率不变的情况下,随着连接数据量的增加,3 种算法的执行效率;(2) 测试在连接数据量不变的情况下,随着重复率的增加,3 种算法的执行效率;(3) 测试在测试 1 中,Pr_LJC 算法中剪枝功能的执行效率;(4) 测试在测试 2 中,Pr_LJC 算法中剪枝功能的执行效率.

测试 1.

这里规定重复率为表数据量的 0.1%,即 6 000 行,采用 Query 1 进行测试,其中,谓词 *A*,*B*,*C*,*D* 用来控制两表的连接数据量以及保证重复度为 0.1%,这里使用如下策略实现(其中,wl 表示单位:万行):

l_part:(*A*,*B*)→{(0,10.6wl),(0,20.6wl),(0,50.6wl),…}

r_part:(*C*,*D*)→{(10wl,20.6wl),(20wl,40.6wl),(50wl,100.6wl),...}.

上述策略能够保证在重复度为 0.1%的前提下,不断增加连接数据量.测试结果如图 9 所示.

Query 1: select *count*(*) from *l_part* inner join *r_part* on *l_part.P_PARTKEY=r_part.P_PARTKEY*

where *l_part.P_PARTKEY>A* and *l_part.P_PARTKEY<B* and

r_part.P_PARTKEY>C and *r_part.P_PARTKEY<D*

分析. 从图 9 中可以看出,在重复度不变的情况下,随着连接数据量的增加,Pr_LJC 算法的执行时间基本不变;而 uPr_uLJ 算法和 uPr_LJ 算法随数据量增加,执行时间显著增长.原因是:由于 Pr_LJC 算法的剪枝(prune)策略,将重复度以外的无用数据块提前去除,其他两种算法完全基于原始数据进行操作.

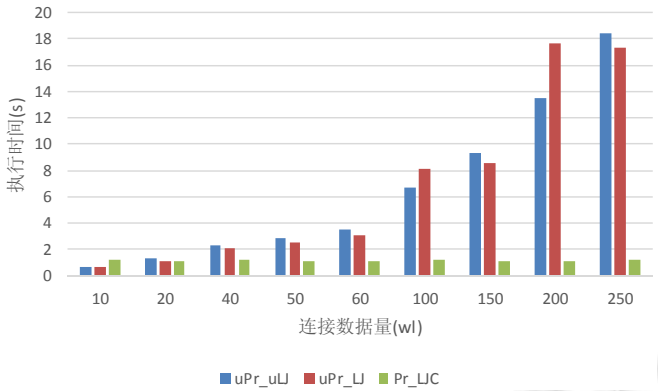

Fig.9 Result of execution efficient comparison of different algorithms under situation of fixing overlap degree and increasing data size

图 9 固定重复度,随着数据量的增加,不同算法执行效率的对比结果

测试 2.

这里规定测试的数据量为 250.6wl,限定重复度以步长 0.6wl 增长.测试语句采用 Query 1,采用与测试 1 类似策略,保证在连接数据量不变的情况下,逐渐增加重复度:

l_part:(*A*,*B*)→{(0,250.6wl),(0.6,251.2wl),(1.2,251.8wl),…}

r_part:(*C*,*D*)→(250wl,500.6wl).

上述策略能够保证在数据量为 250.6wl 的前提下,逐渐增加重复度.测试结果如图 10 所示.

Fig.10 Result of execution efficient comparison of different algorithms under situation of fixing data size and increasing overlap degree

图 10 数据量固定,随着重复度的增加,不同算法执行效率的对比结果

分析. 从图 10 可以看出,在连接数据量固定的情况下,随着重复度的增加,Pr_LJC 算法的执行效率缓慢增加,原因是需要执行连接操作的数据量逐渐增加;uPr_uLJ 算法和 uPr_LJ 算法始终保持较高的执行时间,原因是这两种算法的执行时间不仅与重复度相关,而且依赖于原始连接数据量.

测试 3 和测试 4.

通过在程序中添加时间函数,对 Pr_LJC 中的剪枝功能进行执行效率测试.测试结果如图 11 和图 12 所示.

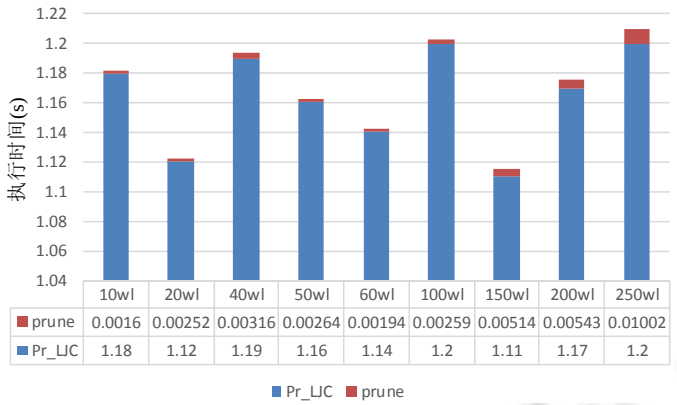

Fig.11 Fixing overlap degree and testing execution efficient of prune in Pr_LJC as the increasing of data size

图 11 固定重复度,测试 Pr_LJC 算法中,prune 功能随数据量增加的执行效率

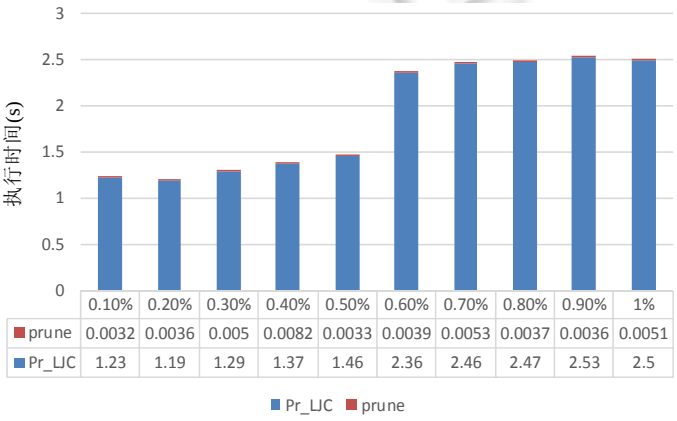

Fig.12 Fixing data size and testing execution efficient of prune in Pr_LJC as the increasing of overlap degree

图 12 固定数据量,测试 Pr_LJC 算法中,prune 功能随重复度增加的执行效率

分析. 从图 11 和图 12 可以看出,在测试 1 和测试 2 两种测试环境下,Pr_LJC 中的剪枝(prune)功能的执行效率对于 Pr_LJC 策略本身来说可以忽略.原因是剪枝功能的实现(详见第 3.1.1 节)完全在内存中进行,并且数据只涉及多个表示区间的数据,因此数据量可以忽略.

7 结论和展望

排序合并连接是数据库系统的一种重要连接方式,大数据时代,由于连接数据量的巨大,特别是分布式环境下,需要考虑网络代价,造成提升连接效率挑战巨大.本文提出了 Pr_PSMJ 策略,在进行实际连接之前,通过构造双边邻接表(BAL)对连接数据进行剪枝,提前去除无用数据块,降低本地代价和网络代价;并通过 BAL 指导本地并行合并连接,最小化数据移动量,有效提升整体连接效率.Pr_PSMJ 策略适应目前大多数合并连接策略.未来需要对切分因子(q)进行更细致的求解,并且对合并区间进行细化,做到更彻底的剪枝以及更健壮的负载均衡;同时,对其他连接策略,如非阻塞式合并排序连接方法等方法进行进一步的研究与优化.

References:

[1] Merrett TH. Why sort-merge gives the best implementation of the natural join. ACM SIGMOD Record, 1983,13(2):39−51. [doi: 10.1145/984523.984526]

[2] Graefe G. Sort-merge-join: An idea whose time has(h) passed? In: Proc. of the 10th Int'l Conf. on Data Engineering. IEEE, 1994. 406−417. [doi: 10.1109/ICDE.1994.283062]

[3] Yang ZK. The architecture of OceanBase relational database system. Journal of East China Normal University (Natural Sciences), 2014,2014(5):141−148,163 (in Chinese with English abstract). [doi:10.3969/j.issn.1000-5641.2014.05.012]

[4] Barthels C, Müleler I, Schneider T, Alonso G, Hoefler T. Distributed join algorithms on thousands of cores. Proc. of the VLDB Endowment, 2017,10(5):517−528. [doi: 10.14778/3055540.3055545]

[5] Albutiu MC, Kemper A, Neumann T. Massively parallel sort-merge joins in main memory multi-core database systems. Proc. of the VLDB Endowment, 2012,5(10):1064−1075. [doi: 10.14778/2336664.2336678]

[6] Balkesen C, Alonso G, Teubner J, Özsu TM. Multi-core, main-memory joins: Sort vs. hash revisited. Proc. of the VLDB Endowment, 2013,7(1):85−96. [doi: 10.14778/2732219.2732227]

[7] Kim C, Kaldewey T, Lee VW, Sedlar E, Nguyen AD, Satish N, Chhugani J, Blas AD, Dubey P. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. Proc. of the VLDB Endowment, 2009,2(2):1378−1389. [doi: 10.14778/1687553.1687564]

[8] Mishra P, Eich MH. Join processing in relational databases. ACM Computing Surveys (CSUR), 1992,24(1):63−113. [doi: 10.1145/128762.128764]

[9] Shapiro LD. Join processing in database systems with large main memories. ACM Trans. on Database Systems (TODS), 1986,11(3):239−264. [doi: 10.1145/6314.6315]

[10] Graefe G. Query evaluation techniques for large databases. ACM Computing Surveys (CSUR), 1993,25(2):73−169. [doi: 10.1145/152610.152611]

[11] Haas PJ, Hellerstein JM. Ripple joins for online aggregation. ACM SIGMOD Record, 1999,28(2):287−298. [doi: 10.1145/304181.304208]

[12] Lin X, Zeng X, Pu X, Sun Y. A cardinality estimation approach based on two level histograms. Journal of Information Science & Engineering, 2015,31(5):1733−1756.

[13] Dittrich JP, Seeger B, Taylor DS, Widmayer P. Progressive merge join: A generic and non-blocking sort-based join algorithm. In: Proc. of the 28th Int'l Conf. on Very Large Data Bases. VLDB Endowment, 2002. 299−310.

[14] Luo G, Naughton JF, Ellmann CJ. A non-blocking parallel spatial join algorithm. In: Proc. of the 18th Int'l Conf. on Data Engineering. IEEE, 2002. 697−705. [doi: 10.1109/ICDE.2002.994786]

[15] Mokbel MF, Lu M, Aref WG. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In: Proc. of the 20th Int'l Conf. on Data Engineering. IEEE, 2004. 251−262. [doi: 10.1109/ICDE.2004.1320002]

[16] Vitorovic A, Elseidy M, Koch C. Load balancing and skew resilience for parallel joins. In: Proc. of the 2016 IEEE 32nd Int'l Conf. on Data Engineering (ICDE). IEEE, 2016. 313−324. [doi: 10.1109/ICDE.2016.7498250]

[17] Wilschut AN, Apers PMG. Pipelining in query execution. In: Proc. of the Int'l Conf. on Databases, Parallel Architectures and Their Applications (PARBASE'90). IEEE, 1990. 562. [doi: 10.1109/PARBSE.1990.77227]

[18] Aslam A, Ansari MS, Varshney S. Non-partitioning merge-sort: Performance enhancement by elimination of division in divide-and-conquer algorithm. In: Proc. of the 2nd Int'l Conf. on Information and Communication Technology for Competitive Strategies. ACM Press, 2016. 1−6. [doi: 10.1145/2905055.2905092]

[19] Perl Y, Itai A, Avni H. Interpolation search—A log log*N* search. Communications of the ACM, 1978,21(7):550−553. [doi: 10.1145/359545.359557]

[20] Andersson A, Mattsson C. Dynamic interpolation search in *o*(log log*n*) time. In: Proc. of the Int'l Colloquium on Automata, Languages & Programming. 1993. 15−27. [doi: 10.1007/3-540-56939-1_58]

[21] Corbett JC, Dean J, Epstein M, Fikes A, Frost C, Furman JJ, Gubarev A, Heiser C, Hochschild P, Hsieh W, Kanthak S, Kogan E, Li H, Lloyd A, Melnik S, Mwaura D, Nagle D, Quinlan S, Rao R, Rolig L, Saito Y, Szymaniak M, Taylor C, Wang R, Woodford D. Spanner: Google's globally distributed database. ACM Trans. on Computer Systems (TOCS), 2013,31(3):251−264. [doi: 10.1145/2491245]

[22] Fan QS, Zhou MQ, Zhou AY. A distributed join algorithm on separated data storage. Chinese Journal of Computers, 2016,39(10):2102−2113 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2016.02102]

[23] Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE. Bigtable: A distributed storage system for structured data. ACM Trans. on Computer Systems (TOCS), 2008,26(2):205−218. [doi: 10.1145/1365815.1365816]

[24] Silberschatz A, Korth HF, Sudarshan S. Database System Concepts. 6th ed., New York: McGraw-Hill, 2010.

[25] Vengerov D, Menck AC, Zait M, *et al*. Join size estimation subject to filter conditions. Proc. of the VLDB Endowment, 2015,8(12): 1530−1541. [doi: 10.14778/2824032.2824051]

[26] Daenen J, Neven F, Tan T, Vansummeren S. Parallel evaluation of multi-semi-joins. Proc. of the VLDB Endowment, 2016,9(10): 732−743. [doi: 10.14778/2977797.2977800]

[27] Chu S, Balazinska M, Suciu D. From theory to practice: Efficient join query evaluation in a parallel database system. In: Proc. of the 2015 ACM SIGMOD Int'l Conf. on Management of Data. ACM Press, 2015. 63−78. [doi: 10.1145/2723372.2750545]

[28] TaoBao. OceanBase. 2018. https://github.com/alibaba/oceanbase

附中文参考文献:

[3] 阳振坤.OceanBase 关系数据库架构.华东师范大学学报(自然科学版),2014,2014(5):141−148,163. [doi: 10.3969/j.issn.1000-5641.2014.05.012]

[22] 樊秋实,周敏奇,周傲英.基线与增量数据分离架构下的分布式连接算法.计算机学报,2016,39(10):2102−2113. [doi: 10.11897/SP.J.1016.2016.02102]

高锦涛(1986－),男,山东青岛人,博士生,主要研究领域为数据管理,分布式查询优化.

杜洪涛(1978－),男,博士,副教授,主要研究领域为海量数据管理,分布式数据库.

李战怀(1961－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为数据库理论与技术,海量数据存储与管理.

刘文洁(1976－),女,博士,副教授,主要研究领域为云计算,大数据处理,海量分布式数据库.