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).

$$\left.\begin{array}{ll}
\text{计算代价} & \text{存储代价} \\
C(uPr_uLJ)=C_fd_1+C_gs_1+C_gmj_1 & S(uPr_uLJ)=S_fd_1+S_gs_1+S_gmj_1 \\
C_fd_1=\sum_{i=1}^{N}(SS_i_IO+C_i_Net+QS_IO) & S_fd_1=\sum_{i=1}^{N}(SS_i_Mem)+QS_Mem \\
C_gs_1=QS_CPU & S_gs_1=QS_Mem \\
C_gmj_1=QS_CPU & S_gmj_1=QS_Mem
\end{array}\right\} \tag{10}$$

- uPr_LJ 算法相关的计算代价和存储代价的计算见公式(11).

$$\left.\begin{array}{ll}
\text{计算代价} & \text{存储代价} \\
C(uPr_LJ)=C_fd_2+C_lps_2+C_lmj_2+C_grm_2 & S(uPr_LJ)=C_fd_2+C_lps_2+C_lmj_2+C_grm_2 \\
C_fd_2=\sum_{i=1}^{N}(SS_i_CPU+SS_i_IO) & S_fd_2=\sum_{i=1}^{N}SS_i_Mem \\
C_lps_2=\sum_{i=1}^{N}SS_i_CPU & S_lps_2=\sum_{i=1}^{N}SS_i_Mem \\
C_lmj_2=\sum_{i=1}^{N}(SS_i_CPU)+\sum_{j=1}^{M}(C_j_Net) & S_lmj_2=\sum_{i=1}^{N}SS_i_Mem \\
C_grm_2=\sum_{i=1}^{N}(C_i_Net)+QS_CPU & S_grm_2=QS_Mem
\end{array}\right\} \tag{11}$$

- Pr_LJC 算法相关的计算代价和存储代价的计算见公式(12).

$$\left.\begin{array}{l}
\text{计算代价} \\
C(Pr_LJC)=C_fd_3+C_lps_3+C_pr_3+C_lmjLJC_3+C_grm_3 \\
C_fd_3=\sum_{i=1}^{N}(SS_i_CPU+SS_i_IO) \\
C_lps_3=\sum_{i=1}^{N}SS_i_CPU \\
C_pr_3=QS_CPU+\sum_{i=1}^{K}C_i_Net \\
C_lmjLJC_3=\sum_{i=1}^{\mu\times N}SS_i_CPU+\sum_{i=1}^{L}C_i_Net, 0<\mu<1 \\
C_grm_3=QS_CPU+\sum_{i=1}^{\mu\times N}C_i_Net \\
\text{存储代价} \\
S(Pr_LJC)=S_fd_3+S_lps_3+S_pr_3+S_lmjLJC_3+S_grm_3 \\
S_fd_3=\sum_{i=1}^{N}SS_i_Mem \\
S_lps_3=\sum_{i=1}^{N}SS_i_Mem \\
S_pr_3=QS_Mem \\
S_lmjLJC_3=\sum_{i=1}^{\mu\times N}SS_i_Mem \\
S_grm_3=QS_CPU+\sum_{i=1}^{\mu\times N}C_i_Net
\end{array}\right\} \tag{12}$$

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).

$$\left.\begin{array}{ll}\text{based on}: & C_fd_2 < C_fd_1 \wedge S_fd_2 < S_fd_1 \wedge \\ & C_lps_2 < C_gs_1 \wedge S_lps_2 < S_gs_1 \wedge \\ & (C_lmj_2 + C_grm_2) \leqslant C_gmj_1 \wedge \\ & (S_lmj_2 + S_grm_2) \leqslant S_gmj_1 \\ infer: & C(uPr_LJ) < C(uPr_uLJ) \wedge S(uPr_LJ) < S(uPr_uLJ)\end{array}\right\} \tag{13}$$

$$\left.\begin{array}{ll}\text{based on}: & C_fd_3 = C_fd_2 \wedge S_fd_3 = S_fd_2 \wedge \\ & C_lps_3 = C_lps_2 \wedge S_lps_3 = S_lps_2 \wedge \\ & C_lmjLJC_3 < C_lmj_2 \wedge S_lmjLJC_3 < S_lmj_2 \\ & C_grm_3 < C_grm_2 \wedge S_grm_3 < S_grm_2 \\ & C_pr \text{ and } S_pr \text{ very small} \\ infer: & C(Pr_LJC) < C(uPr_LJ) \wedge S(Pr_LJC) < S(uPr_LJ)\end{array}\right\} \tag{14}$$

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.

© 中国科学院软件研究所 http://www.jos.org.cn

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.

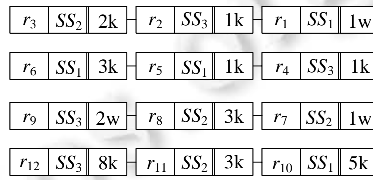


Fig.6 Instance of left part of BAL

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.

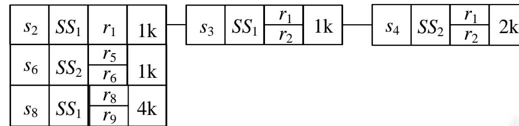


Fig.7 Instance of right part of BAL

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.

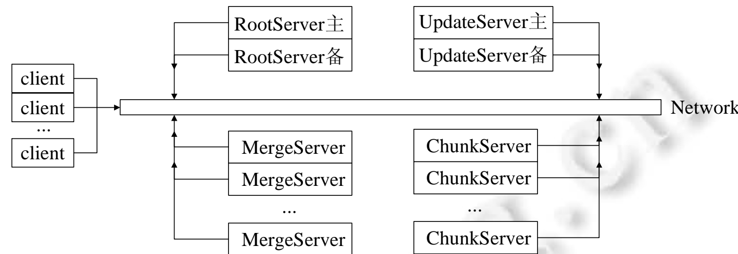


Fig.8 Architecture of OceanBase

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:$(A,B)\rightarrow$\{(0,10.6wl),(0,20.6wl),(0,50.6wl),…\}

r_part:$(C,D)\rightarrow$\{(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.

© 中国科学院软件研究所 http://www.jos.org.cn

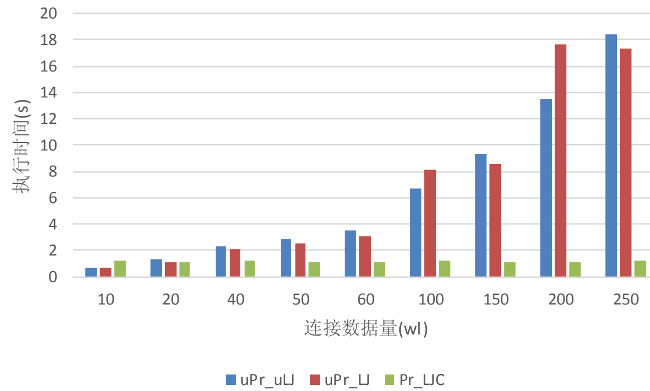


Fig.9 Result of execution efficient comparison of different algorithms under situation of fixing overlap degree and increasing data size

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.

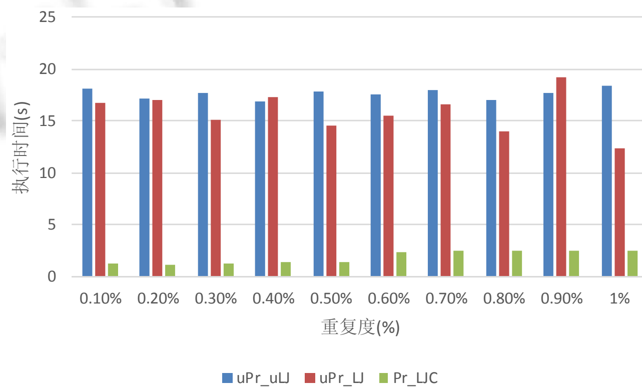


Fig.10 Result of execution efficient comparison of different algorithms under situation of fixing data size and increasing overlap degree

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.

© 中国科学院软件研究所 http://www.jos.org.cn

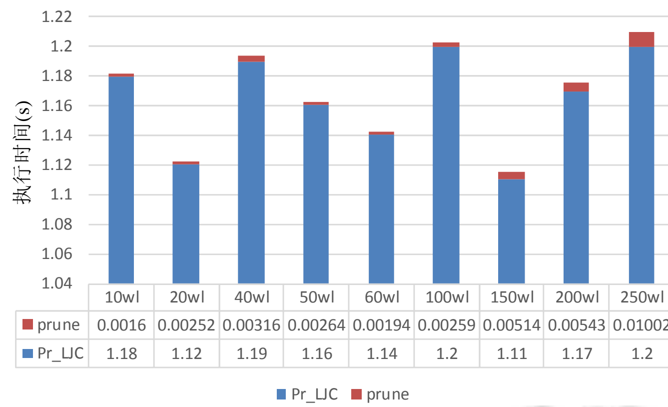


Fig.11 Fixing overlap degree and testing execution efficient of prune in Pr_LJC as the increasing of data size

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

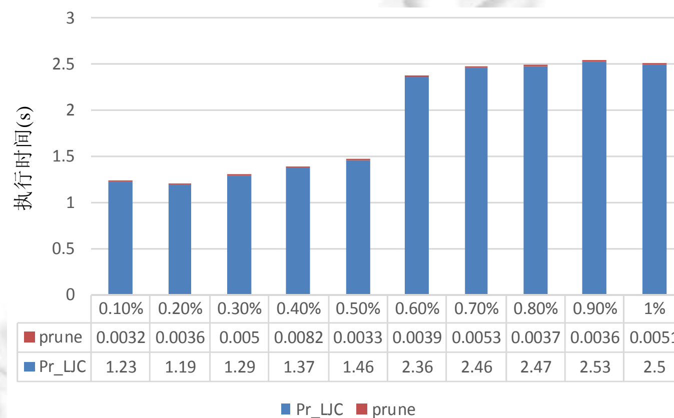


Fig.12 Fixing data size and testing execution efficient of prune in Pr_LJC as the increasing of overlap degree

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*(loglog*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]

© 中国科学院软件研究所 http://www.jos.org.cn

[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]

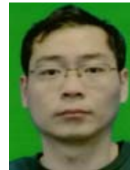
[28] TaoBao. OceanBase. 2018. https://github.com/alibaba/oceanbase

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.

© 中国科学院软件研究所 http://www.jos.org.cn