Multi-Agent Join

Vahid Ghadakchi
Oregon State University
ghadakcv@oregonstate.edu

Mian Xie
Oregon State University
xiemia@oregonstate.edu

Arash Termehchy
Oregon State University
termehca@oregonstate.edu

Michael Burton
Oregon State University
burtomic@oregonstate.edu

Abstract

It is crucial to provide real-time performance in many query workloads, such as interactive and exploratory data analysis. In these settings, users need to view a subset of query results or a progressive presentation of the entire results quickly. Nevertheless, it is challenging to deliver such results over large dataset for common relational binary operators, such as join. Join algorithms usually spend a long time on scanning and attempting to join parts of relations that may not generate any result. Current solutions to this problem usually require lengthy and repeating preprocessings, which are costly in general settings and may not be possible to do in some cases, such as interactive workloads or evolving datasets. Also, they may support restricted types of joins. In this paper, we outline a novel approach for achieving efficient progressive join processing in which the scan operator of the join learns online and during query execution the portions of its underlying relation that might satisfy the join condition and use them in the join. We further improve this method by an algorithm in which both scan operators collaboratively learn an efficient join execution strategy. We also show that this approach generalizes traditional and non-learning methods of join.

1 Introduction

It is crucial to provide real-time performance for queries over large data in many settings, such as interactive and exploratory data analysis or online query processing Carey and Kossmann (1997, 1998); Hellerstein et al. (1997); Haas and Hellerstein (1999); Li et al. (2016). The recent pandemic showed an urgent need for analyzing the ginormous data quickly to model and forecast public health concerns in real-time for Disease Control and Prevention (2021); dj patil (2020). For example, it is vital that epidemiologists interactively test various queries about how a novel virus spreads over numerous and continually evolving case reports efficiently to recommend effective public policies as fast as possible. In these settings, it is desirable to provide subsets of query results quickly to reduce the time that users spend on data exploration and interaction. For instance, in an interactive environment, an epidemiologist may want to view subsets of answers to her query fast to design her next query according to these answers quickly. In these settings, users might also want to view results of their queries progressively without any long (initial) delays Haas and Hellerstein (1999); Li et al. (2016); Jermaine et al. (2005). This enables users to view and investigate the results without long waiting periods and as the results are being produced. This capability reduces users' waiting time and speeds up their data analysis tasks. For example, it is important to present the query results without long delays to the epidemiologist, so she can investigate them quickly as they become available. Using this method, users can stop the execution quickly and as soon as they have enough information.

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

It is, however, challenging to find quickly subsets or the entire results of many binary relational operators, such as *join*, over relations with many tuples. To execute these operators, the relational data system has to check numerous if not all pairs of tuples from both input relations to find the ones that satisfy the operator's predicate. In many join queries, a substantial majority of these pairs do not satisfy the join predicate. As the system often does not know the pairs of tuples that satisfy the predicate, it may spend a long time to check numerous pairs without returning any or very few results. Also, the information of input relations often stored on secondary storage and accessing them takes lengthy I/O accesses. Hence, returning subsets or the entire join results might be very time-consuming and involve long delays. Thus, users may have to wait for a long time to view query results. This has become more challenging due to the rapid growth and frequent evolution of datasets.

To address this problem, researchers have proposed algorithms that progressively read tuples from base relations and maintain them in main memory to improve I/O efficiency of the query execution, e.g., *Ripple Join* Haas and Hellerstein (1999). These methods, however, quickly run out of main memory over large relations before generating sufficiently many results Jermaine et al. (2005). Some methods build auxiliary data structures, e.g., indexes, over both relations to locate tuples that generate results quickly Li et al. (2016). Most users, however, cannot afford to wait for the time-consuming preprocessing steps of building these data structures, which are often repeated whenever the data evolves. The indexes needed for a query workload are usually determined by experts or pre-trained models Chaudhuri and Narasayya (2007); Aken et al. (2021). Most normal users, such as epidemiologists, do not have the expertise to guide index building effectively. There might not also be sufficient training data to learn an accurate model of the query workload to guide these preprocessing steps. Moreover, current guiding models usually assume that the distribution of query workload is fixed over time Chaudhuri and Narasayya (2007); Aken et al. (2021). Users, however, increasingly produce non-stationary workloads particularly in interactive settings McCamish et al. (2018). Also, indexes often take substantial storage and update overheads. Users might not have enough resources to build and maintain indexes for many queries.

Sort- and hash-based join algorithms produce the entire results of a join query efficiently but they need to reorganize the input relations in time-consuming steps before generating any result Garcia-Molina et al. (2009). Researchers have also extended these algorithms for progressive query computation Jermaine et al. (2005). There methods, however, support only a limited types of queries, e.g., joins with equality conditions Garcia-Molina et al. (2009). Due to the popularity of statistical learning and inference over large data, join predicates increasingly contain complex conditions, such as user-defined functions or inequality conditions. They might also need a significant amount of main memory to generate results efficiently, which might not be available, e.g., progressive sort-based keeps considerable portions of both relations in main memory Jermaine et al. (2005).

Toward realizing the vision of delivering real-time insight over large data, we propose a novel approach to binary join processing in which scan operators of the join collectively *learn online and during query execution* to return results quickly. The query plan for joining two relations is typically modeled as a tree with two scan operators as leaves and a join operator as its root. During query execution, each scan reads and sends tuples to the join operator and is informed of the success of its sent data in producing results by the join operator. Every scan operator has a strategy of reading tuples and sending them to the join operator in the query. Based on this feedback, the scan operators quickly learn the portions of data that are most likely to generate joint tuples and improves the efficiency of its strategy. Hence, they read and send portions of the data that generate more results earlier in query execution and avoid or postpone sending fragments of the data that might not contribute toward producing joint tuples. This method produces subsets of join results quickly. Since the scan operators learn the most promising portions of relations online and by performing the join, our method does not involve long (initial) delays to produce results. This approach does not require any upfront data preparation, offline training data, or precomputed data structures and their corresponding delays, overheads, and tuning. It can be used for both stationary and non-stationary query workloads. Each scan treats its join operator and its condition as a black box, therefore, it can learn efficient execution strategies joins other than the ones with equality conditions. This method is naturally implemented within the pipelining framework of query processing in current relational data systems.

Nevertheless, it is challenging to accurately learn efficient query processing strategies over large data and produce query results quickly at the same time. Each scan operator should establish a trade-off between producing fresh results based on its current information and strategy, i.e., *exploitation*, and searching for more efficient strategies, i.e., *exploration*. Moreover, as each relation may contain

numerous tuples, each scan operator may have to explore many possible strategies to find the efficient ones. This becomes more challenging by the usual restrictions on accessing data on the secondary storage. For example, in the absence of any index, a scan operator has to examine tuples (blocks) on disk sequentially. Current online learning algorithms, however, often assume that each learner has a random access to all its alternatives. In this paper, we leverage tools from the area of online learning and devise algorithms that address the aforementioned challenges efficiently.In particular, we show that it will improve the time of producing subsets of join results, if both scan operators of join learn and adapt their strategies online. One might extend our framework and methods to some other types of binary relational operators, such as intersection. However, due to popularity of joins, in this paper, we focus on and report empirical studies for join queries.

2 Framework

We present the framework that models processing of a join query method as online collaborative learning of its operators. Our framework extends to other binary operators, such as intersection. Nonetheless, we restrict our attention in this paper to binary joins and leave detailed explanation and instantiating our framework for other types of queries as future work.

Agents and Actions. Relational data Systems usually model a query as a tree of logical operators called *logical query plan* Garcia-Molina et al. (2009). In a join query, the leaf nodes are scan operators that read information form the secondary storage in blocks (pages). The tuples are pipelined from scans to the root operator, i.e., join, as the join operator calls its children's API to get fresh tuples. Each *iteration* (or *round*) of processing the query starts with the call from the join operator. In our framework, each scan operator in the query plan is a learning *agent*. The *actions* of an agent as the set of its available activities in each round. The set of actions of each scan operator is the set of blocks in its base relation. As motivated in the preceding section, in this paper, we assume that both relations to be joined are large and none of them are small enough to fit into available main memory.

Reward. The *reward* of an action in each round of the query execution is the total number of joint tuples produced during that round. The join operator shares this reward with its child scan operators immediately after attempting to join their recently sent blocks. As we assume that the relations are stored on the secondary storage, each agent has to perform some I/O access(es) to read block(s) from the secondary storage. Since the dominating cost of performing joins is the time to perform I/O access, ideally, each scan operator may receive some reward in each round so that its effort of reading a block from disk produces some answers and does not go to waste. This increases the I/O efficiency of query execution. The *history* of a scan operator O at round t denoted as $H^O(t)$, is the sequence of pairs (a_i, r_i), $0 \leq i \leq t-1$, where a_i and r_i are the action and the reward of the operator O at round i of the join.

Strategies. The *strategy* or policy of an operator O at round t is a mapping from $H^O(t)$ to the set of its available actions. A strategy is essentially the execution algorithm of its (logical) operator. An operator might follow a *fixed strategy* that do not change in the course of query execution. Current query operators often follow fixed strategies. For example, the scan operator for the outer relation in the (block-based) nested loop join plan follows a fixed strategy of sending the next block of the relation stored on the disk whenever requested by its parent join operator. The scan operator for the inner relation follows a similar fixed strategy. An operator may use an *adaptive strategy* and choose actions in each round based on the actions' rewards and performance in their previous rounds using the history up to the current round. In particular, if the underlying relations contain sufficiently many blocks, i.e., a join has sufficiently many rounds, a scan operator may achieve a higher long-term reward by modifying its strategy during the query processing. For example, scan operators for the outer and inner relations in the nested loop join may use their history to estimate and send the blocks that are more likely to produce new joint tuples instead of the ones that may not lead to any results. Using the history of the join, a scan operator may learn that block b_1 in its base relation joins with significantly more tuples of the other relation than block b_2, i.e., b_1 is more rewarding than b_2. Thus, if it reads and sends tuples from b_1 to the join operator more often than b_2, it may cause the query to produce more joint tuples by performing the same number of I/O accesses. As the query execution progresses, the operators may learn and improve their estimate and the effectiveness of their strategies using the performance of their actions in preceding rounds. Our framework generalizes current approaches to query processing as operators that follow fixed strategies. Hence, it supports using both traditional and adaptive strategies for query operators.

Learning Strategies. Since the rewards of actions are not known at the start of the query processing, an operator has to learn these rewards while executing the query. Thus, the operator may initially

explore a subset of available actions to find the reasonably rewarding actions quickly. Subsequently, the operator may *exploit* this knowledge and use the most rewarding actions according to its investigations so far to increase the short-term overall reward or *explore* actions that have *not* been tried to find actions of higher rewards with the goal of improving the total reward in the long-run. For example, consider the join of relations R and S where the scan over S uses a fixed strategy of sending a randomly chosen block in each round. The scan over R modifies its strategy in each round based on the information in the preceding rounds to read and send blocks that generate most joint tuples to the join operator. The scan over R may initially send a few randomly chosen blocks to its parent join operator to both produce some joint tuples and estimate the average joint tuples generated from each block in the subset, i.e., its reward. In the subsequent rounds, the scan operator on R may send the blocks with highest rewards so far, i.e., exploit, or pick other blocks that have not been tried before with the hope of finding ones with higher estimated reward than current ones, i.e., explore. An important challenge in online learning is to find a balance between exploration and exploitation.

Overall Objective Function. A query processing algorithm should return the results of a query as fast as possible, therefore, each operator should maximize its total reward using fewest possible rounds. As the datasets are often very large, it is desirable to deliver the results progressively where users see and investigate earlier tuples quickly while the system executes the query and delivers the rest of the results Carey and Kossmann (1997); Haas and Hellerstein (1999); Hellerstein et al. (1997). The user may stop the query execution as soon as she receives a sufficient amount of information, e.g., sufficiently many tuples, or let it run until completion. This is particularly useful in interactive or exploratory workloads where users need to know at least an estimate of the results very fast. Moreover, many analyses over large data may be satisfied by a sufficiently large subset of query results Carey and Kossmann (1997); Haas and Hellerstein (1999); Hellerstein et al. (1997). To quantify the overall objective of query processing, One may select a metric, such as *discounted (weighted/geometric) average of delays*, that is biased toward faster generation of early results. It measures the users' waiting time for receiving both a sample of and the full query result. More precisely, the discounted weighted average of delays is defined as $\sum_{i=1}^{l} \gamma^i t_i$ where $0 < \gamma < 1$, t_i is the time to generate the ith result, and l is set to the number of desired results. The faster a query operator learns an efficient strategy during execution, the larger the value of objective function is.

3 Single Scan Learning

In this section, we investigate learning for only one of the scan operator in the join of R and S, namely R-scan. We first assume that S-scan has a fixed strategy of reading a randomly chosen block from S and sending it to the join operator, i.e., *random strategy*. It is shown that in the absence of any order, e.g., sorted relation, sequential scan, i.e., heap-scan, simulate random sampling effectively Haas and Hellerstein (1999). Thus, we assume that S-scan implements the random strategy using the sequential scan over S. On the other hand, R-scan aims at learning the rewards of blocks in R accurately and quickly and joining blocks of R with S in a decreasing order of their rewards. This way, the join operator progressively produces results efficiently and significantly reduces the users' total waiting time to view query results as explained in Section 2. As R-scan does *not* know the reward of blocks of R prior to join executing, its learning should take place during join processing.

Our algorithm constitutes of a series of *super-rounds*. In each super-round, the algorithm learns and selects the most rewarding block from all blocks in R that have not yet been selected and uses the selected block to generate join results. It also produces results while learning the most rewarding block. The algorithm continues to the next super-round until it generates a given number of tuples or the complete results based on users' input.

3.1 Learning Most Rewarding Block

To estimate the most rewarding block in R, we use a *many-armed MAB* algorithm called M*-Run*, that effectively estimate actions with relatively high reward over infinitely many actions by exploring a sufficiently large random subset of them Berry et al. (1997). We first introduce an exploration technique over R called N*-Failure*. We define each *round* as the join of a block of R and a block S. Using N-Failure, initially, R-scan and S-scan (sequentially) read a new block form their relations and send these blocks to the join operator to join. In each subsequent round, S-scan reads a fresh block from S. R-scan, on the other hand, does not read any block and keeps sending its current block in main memory to the join operator if this block has produced at least one joint tuple during the last N rounds. Otherwise, it reads the next block of R using sequential scan and sends it to the join operator. The reward of each block of R is the number of joint tuples it produces during its

N-Failure exploration. For each block of R with non-zero reward, R-scan maintains its address to reward mapping in a reward table stored in the main memory.

R-scan performs N-Failure for every block of R until it either scans M blocks of R or finds a block of R that produces at least one join tuple in each of last M rounds. At this point, R-scan and S-scan *fully exploit* the most rewarding block in R by performing a full join of this block and entire S. That is, R-scan picks the block with the highest reward from the reward table, reads it from disk using random access, and sends it to the join operator. S-scan resets its sequential scan form the beginning of S, reads S block by block and sends each block to the join operator. As the most rewarding block of R may have already been joined with a sequence of blocks in S during its N-Failure turn, R-scan keeps track of this range of block numbers for each block in the reward table. If the reward signal is sparse, e.g., the join is very selective, in some super-rounds, the rewards of all examined blocks may be zero. In this case, R-scan picks its last scanned block that is already in main memory, i.e., Mth block, for the full join. It stores the information of this block in reward table to track all blocks that have been fully exploited.

3.2 Subsequent Super-Rounds

After each super-round, R-scan excludes the most rewarding block from its list of available actions. In each subsequent super-rounds, R-scan resumes its sequential scan from the last position that it was stopped. It reads and sends the next unread block of R to the join operator, explores its reward using N-failure technique, and adds its reward and address to the reward table in main-memory if its reward is non-zero. S-scan will continue its sequential scan of S in each step of this N-failure exploration. M-run accesses blocks sequentially and evaluates each accessed block reward using N-failure exploration technique only once. Because R-scan has the reward information of a new set of M blocks, after reading only one new block, it has enough information to estimate a new most rewarding block by selecting the block with most reward in the reward table. Thus, in each super-round after the first one, R-scan estimates the new most rewarding block by reading only one new block from R. It then accesses the selected block using its address in the reward table.

4 Collaborative Scans

In m to n joins of R and S, some blocks of S may have substantial reward. Hence, it may reduce the response time of the join if R-scan and S-scan both use learning strategies. In this setting, each scan operator should both *learn the reward of its blocks* and *provide randomly sampled blocks* for the other scan so they both learn the rewards of their own blocks. This may double the number of I/O accesses or data processing effort by each scan during learning and slow down join processing significantly.

One approach to address this problem is to interleave and combine learning and sampling such that the set of explored blocks of one scan is also a random set of blocks from its base relation. That is, each scan may view each block as both an action of itself and also a sample from the environment for the other scan in the join. Each explored action in M-Run is in fact a random sample of available actions. Thus, if every scan operator uses M-Run learning method, in each exploration, it sends to the join operator both one action of its available actions, i.e., blocks, and a random sample of blocks of its underlying relation. Hence, each scan can both learn the reward of its own actions and at the same time enable the other scan to measure the rewards of its current actions by observing a random sample from its environment.

More precisely, in each super-round, one scan performs an N-Failure exploration and the other one performs a sequential scan. They will switch strategies for the subsequent super-round. Each scan that performs N-Failure collects rewards and other relevant information of its explored blocks as explained in Section 3. After reaching to its Mth explore block, each scan decides its most rewarding block and fully joins with with the other relation. In the subsequent super-rounds, the scan operators follow the algorithm described in Section 3. The join stops after getting the desired number of tuples.

Because each scan switches turns between M-Run and random strategies, it may sequentially read some blocks during its random strategy and may *not* resume its next N-Failure exploration immediately after the block that it explored in its last N-Failure exploration. Hence, it may skip doing N-Failure for some blocks. To ensure that each scan explores every block, it has to *go back* to the position after its last N-Failure. Nonetheless, the implementations of sequential scan in current relational data system implementations usually starts from the beginning of the relation. We have observed this in particular in PostgreSQL, which we have used to implement our algorithms. Thus, to resume the sequential scan from the last block for which the scan has done N-Failure, it has to restart

the scan from the beginning and read and skip potentially many block, especially towards the middle and end of the algorithm. To save I/O access time, in our current implementation, our scans do not go back on disk and continue their sequential scan when they switch turns. Therefore, our current implementation of the case where both scan operators learn may *not* produce the full join results.

5 Initial Empirical Results

We evaluate our proposed methods and the comparable methods explained in Section 1 that do not require lengthy preprocessing to create auxiliary data structure, e.g., index, and are not limited to certain types of joins, e.g., only equijoin Jermaine et al. (2005); Li et al. (2016). We compare our methods to *Ripple Join* Haas and Hellerstein (1999) and block-based nested loop join (BNL) Garcia-Molina et al. (2009) that satisfy the aforementioned properties. Ripple-Join quickly runs out of main memory in almost all settings before generating answers as explained in Section 1.

We use TPC-H benchmark, *www.tpc.org/tpc*, to generate the queries and databases for our experiments. We use the $scale = 1$ to generate a TPC-H database of size 1 GB. We did not use larger values as it takes a very long time, e.g., more than a day, for the BNL to process some queries over larger versions of TPC-H database. We have used the following M-N joins from TPCH workload: Q_9:$partsupp \bowtie_{partkey} lineitem$ and Q_{11}: $orders \bowtie_{o\_orderdate=l\_shipdate} lineitem$

One of the parameters that impacts the join processing time is the distribution of the join attribute values; more specifically their skewness. We evaluate the query run-time over different versions of TPC-H database with different degrees of skewness in the join attributes. Our experiments use Zipf distributions with z values ranging between $[0, 0.5, 1, 1.5, 2]$ wherein $z = 0$ will result in uniform distribution and for $z \geq 0.5$, the distribution becomes more skewed as the value of z grows. Moreover, the results of our experiments using $z = 0.5$ are very similar to the ones with $z = 0$. Hence, we do not report the results of experiments with $z = 0.5$ in the this section. We run each query 5 times and report the average time across these 5 times. Each reported response time of BNL is the average of the response times of two different sets of runs with each relation as the outer one.

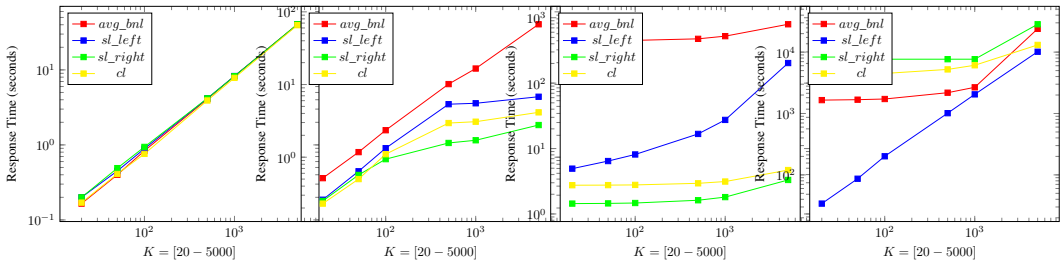
We have implemented our proposed methods and BNL inside PostgreSQL 11.5 database management system. We have performed our empirical study on a Linux server with Intel(R) Xeon(R) 2.30GHz cores and 500GB of memory. We have used query $Q_{12} : orders \bowtie_{orderkey} lineitem$ to train the hyper-pramters of the algorithms, e.g,. N.

Figure 1 and Figure 2 show the response times of BNL, single scan learning runs (denoted as *sl_left* and *sl_right*), and collaborative scans (denoted as *cl*) over Q_9 and Q_{11}, respectively. The response times of the collaborative scans runs are generally between single scan learning on either relations. In each setting, one of the relations may contain the most rewarding blocks, therefore, one of the single scan learning runs delivers the most efficient results. Nonetheless, it is *not* clear which one of scan operators should use a learning strategy to deliver the most efficient results without actually running the query. The collaborative scans approach provides a middle-ground between the two possible configurations for single scan and is generally more efficient than the slowest configurations. All these learning algorithms are at least as efficient as BNL where the data is uniform and more efficient than BNL in almost all other settings. The differences between the learning algorithms and BNL become generally more significant as the skewness parameter z increases. We again observe the same exception in Q_9 at $z = 2$ and as we discussed in the preceding section. It is very hard to learn rewarding blocks on this query as the rewarding blocks are very rare due to highly skewed join attributes on both relations. Our empirical results using weighted discounted average provide a similar outcome for different methods over our workload.

References

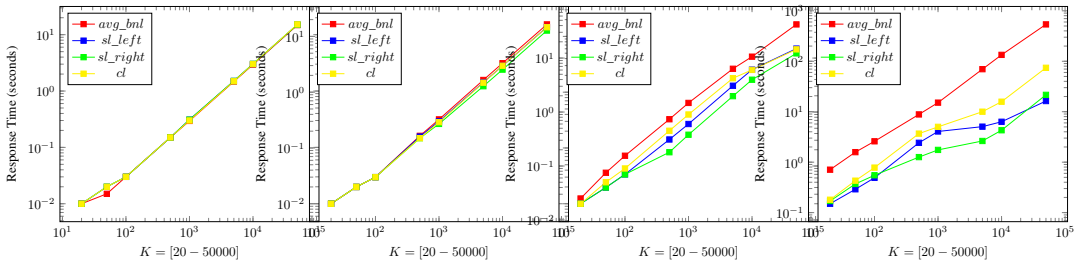
Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Billian, and Andrew Pavlo. 2021. An Inquiry into Machine Learning-based Automatic Configuration Tuning Services on Real-World Database Management Systems. *Proc. VLDB Endow.* 14, 7 (2021), 1241–1253.

Donald A. Berry, Robert W. Chen, Alan Zame, David C. Heath, and Larry A. Shepp. 1997. Bandit problems with infinitely many arms. *The Annals of Statistics* 25, 5 (1997), 2103 – 2116. `https://doi.org/10.1214/aos/1069362389`



(a) z = 0 (b) z = 1 (c) z = 1.5 (d) z = 2

Figure 1: Response times of generating join samples using Collaborative Scans (cl), Single Scan Learning (sl_left and sl_rigth), and BNL over Q_9.



(a) z = 0 (b) z = 1 (c) z = 1.5 (d) z = 2

Figure 2: Response times of generating join samples using Collaborative Scans (cl), Single Scan Learning (sl_right and sl_left), and BNL over Q_{11}.

Michael J. Carey and Donald Kossmann. 1997. On Saying "Enough Already!" in SQL. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997*, Joan Peckham (Ed.). ACM Press, Tucson, Arizona, USA, 219–230. https://doi.org/10.1145/253260.253302

Michael J. Carey and Donald Kossmann. 1998. Reducing the Braking Distance of an SQL Query Engine. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998*, Ashish Gupta, Oded Shmueli, and Jennifer Widom (Eds.). Morgan Kaufmann, New York City, New York, USA, 158–169.

Surajit Chaudhuri and Vivek R. Narasayya. 2007. Self-Tuning Database Systems: A Decade of Progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases, September 23-27, 2007*, Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold (Eds.). ACM, University of Vienna, Austria, 3–14.

dj patil. 2020. 6 lessons learned to get ready for the next wave of COVID. https://medium.com/@dpatil/6-lessons-learned-to-get-ready-for-the-next-wave-of-covid-ee595766d4cb

Centers for Disease Control and Prevention. 2021. CDC Stands Up New Disease Forecasting Center. https://www.cdc.gov/media/releases/2021/p0818-disease-forecasting-center.html

Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2009. *Database systems - the complete book (2. ed.)*. Pearson Education, New Jersey, USA.

Peter J. Haas and Joseph M. Hellerstein. 1999. Ripple Joins for Online Aggregation. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999*, Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh (Eds.). ACM Press, Philadelphia, Pennsylvania, USA, 287–298. https://doi.org/10.1145/304182.304208

Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. 1997. Online Aggregation. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997*, Joan Peckham (Ed.). ACM Press, Tucson, Arizona, USA, 171–182. https://doi.org/10.1145/253260.253291

Chris Jermaine, Alin Dobra, Subramanian Arumugam, Shantanu Joshi, and Abhijit Pol. 2005. A Disk-Based Join With Probabilistic Guarantees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, June 14-16, 2005*, Fatma Özcan (Ed.). ACM, Baltimore, Maryland, USA, 563–574. https://doi.org/10.1145/1066157.1066222

Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation via Random Walks. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, San Francisco, CA, USA, 615–629. https://doi.org/10.1145/2882903.2915235

Ben McCamish, Vahid Ghadakchi, Arash Termehchy, Behrouz Touri, and Liang Huang. 2018. The Data Interaction Game. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. ACM, New York, NY, USA, 83–98. https://doi.org/10.1145/3183713.3196899