A New Framework for Join Product Skew

Foto Afrati[1], Viktor Kyritsis[1], Paraskevas Lekeas[2], Dora Souliou[1]

[1] National Technical University of Athens, Athens, Greece,
{afrati, vkyri}@softlab.ece.ntua.gr, dsouliou@mail.ntua.gr
[2] Department of Applied Mathematics, University of Crete, Herakleio, Greece,
plekeas@tem.uoc.gr

Abstract. Different types of data skewness can result in load imbalance in the context of parallel joins under the shared nothing architecture. We study one important type of skewness, join product skew (JPS). A static approach based on frequency classes is proposed which takes for granted the data distribution of join attribute values. It comes from the observation that since the join selectivity can be expressed as a sum of products of frequencies of the join attribute values, an appropriate assignment of join sub-tasks, that takes into consideration the magnitude of the frequency products can alleviate the join product skew. Motivated by the aforementioned ascertainment, we propose an algorithm, called Handling Join Product Skew (HJPS), to handle join product skew.

Key words: Parallel DBMS, join operation, join selectivity, data distribution, data skew, load imbalance, shared nothing architecture

1 Introduction

The expansion of data in higher volumes as well as the increase complexity of relational database queries make the performance issue a critical element in the design of database systems. The limited potentials of centralized database systems in terms of storage and processing power has led to the advent of parallel database management systems (PDBMS), which allow the process of relational database operations in parallel. Parallelism either by streaming the output of one operator into the input of another one or by splitting an individual operator into many independent operators can lead to faster evaluations of complex queries.

Among the relational operations, join is the most expensive one. Especially, natural join is the most popular form of join operation. It is defined as a binary operation between two relations, say R and S, that include the attributes X and Y respectively in their schema definition. The output of the natural join is the set, whose each element is the concatenation of a tuple r belonging to R and a tuple s belonging to S under the condition that the values of the attributes X and Y in the tuples r and s respectively are equal. The attributes X and Y that participate in the natural join are called join attributes while the equality condition is denoted by the join predicate $R.X = S.Y$. $R \overset{R.X=S.Y}{\bowtie} S$ stands for the natural join of relations R and S on join attributes X and Y. However, for simplicity we use the notation $R \bowtie S$ in this paper.

Sort-merge join and hash-join constitute the two major operators join operator algorithms for the computation of the natural join $R \bowtie S$, that are subject to parallel execution. In this paper we focus on the hash-based join since it has linear execution cost (contrary to sort-merge join), and it performs better in the presence of data skew [4] as well. Especially, we propose a new model for handling join product skew, which is a specific type of skewness, in the context of the shared-nothing architecture [12]. According to this architecture, the computational nodes that constitute a PDBMS are distinguished into two categories: database processors (or parallel units) and control processors. Each computational node has its own memory and CPU and independently accesses its local disks. A database processor is provided with the ability to perform locally relational operations. We will assume that all the database processors have identical configuration. On the other hand the functionality of a control processor is twofold. It provides an interface to users in order to submit their queries, expressed in a programming language, e.g., SQL, and it also sends the database requests to the database processors. All the processors are exchanging messages by using the underlying interconnection network.

The hash-based join processing of two relations in a PDBMS adopting the shared-nothing setup is separated into three phases. In the first phase, which can be regarded as a pre-processing step since its functionality bears no direct relation to the join operation, both relations are fully declustered across the database processors. Provided that relations of large data volumes are considered in the join operation $R \bowtie S$, full declustering is proven to be the best placement strategy for parallel shared-nothing database systems [11]. In general, the declustering attribute of each relation is different from the join attribute. Each relation is horizontally partitioned among the databases processors by applying a partition function (or defining ranges of values) on the declustering attribute.

What follows in the join processing is the redistribution of the partitioned tuples. During the redistribution phase, each database processor applies a common hash function h (which is different from the function used in declustering) on the join attribute value for its local fragments of relations R and S. The hash function h ships any tuple belonging to either relation R or S, with join attribute value b_i to the $h(b_i)$-th database processor. Generally speaking, according to the given definition of the hash function, tuples with different values of join attribute are likely to be shipped to the same database processor. Obviously, if the target database processor determined by the hash function coincides with the database processor, where initially a given tuple is placed, there is no need to ship this tuple. However, in general, this is not the case. At the end of the redistribution process both relations are fully partitioned into disjoint fragments.

Finally, after the completion of the redistribution phase, each database processor p performs locally an equi-join operation between its fragments of relations R and S, denoted by R^p and S^p respectively. The sets R^p and S^p contain tuples that belong to relations R and S respectively, and the hash value of the join attribute in every tuple $t \in R^p \cup S^p$ equals to p. The database optimizer of each database processor chooses the most cost-effective way to execute the

local join operation. It is assumed that the join operation is computed locally by the conventional hash-based algorithm. The joined tuples are kept locally in each database processor instead of being merged with other output tuples into a single stream.

Skewness is identified as one of the major factors that affects the effectiveness of the parallel join [10]. It is perceived as the variance in the response times of the database processors that participate in the parallel computation of the join operation. [13] provides a classification of data skewness into four categories. In sum, the tuple placement skew (TPS) is related to the first phase. It may happen when the number of tuples belonging to a relation varies across the database processors after its declustering. A wise choice of the function that renders the initial placement of tuples in the database processors confines the effect of this specific type of skewness. As to selectivity skew (SS), it occurs when the application of the qualification condition(s), specified in the WHERE clause of the query, leads to a widely varying number of tuples across the database processors. Thus, this type of skewness is query-dependent.

The two remaining types of skewness, redistribution skew (RS) and join product skew (JPS), are concerned to be critical in achieving high performance and scalability in the context of natural join operations. Redistribution skew can be observed after the end of the redistribution phase. It happens when at least one database processor has received large number of tuples belonging to a specific relation, say R, in comparison to the other processors after the completion of the redistribution phase. This imbalance in the number of redistributed tuples is due to the existence of naturally skewed values in the join attribute. Redistribution skew can be experienced in a subset of database processors. It may also concern both the relations R and S (double redistribution skew). Join product skew occurs when there is an imbalance in the number of join tuples produced by each database processor. [11] points the impact of this type of skewness to the response of the join query.

Query load balancing in terms of the join operation is very sensitive to the existence of the redistribution skew and/or the join product skew. Especially, join product skew deteriorates the performance of subsequent join operation since this type of skewness is propagated towards the query tree. In this paper we address the issue of join product skew. We introduce the notion of frequency classes, whose definition is based on the product of frequencies of the join attribute values. Under this perspective we examine the cases of homogeneous and heterogeneous input relations. We also propose a new static algorithm, called HJPS (Handling Join Product Skew) to improve the performance of the parallel joins in the presence of this specific type of skewness. Finally, we consider the case of the chain join. In order to apply our approach, we give an analytical formula about the size of the result set expressed as a product of join selectivities. **Various dynamic and static techniques and algorithms have been proposed to confine the impact of the join product skew.**

The rest of this paper is organized as follows. In section 2 we illustrate the notion of division of join attribute values into classes of frequencies by means of

two generic cases. In section 3 an algorithm that helps in reducing join product skew effects is proposed. Section 4 discusses the related work and section 5 concludes the paper. Finally, a proof of an analytical formula concerning the size of the result set for the chain join is exhibited in appendix.

2 Two Motivating Examples

We will assume the simple case of a binary join operation $R_1(A, B) \bowtie R_2(B, C)$, in which the join predicate is of the form $R_1.B = R_2.B$. The m discrete values $b_1, b_2, \ldots, b_m$ define the domain D of the join attribute B. Let $f_i(b_j)$ denote the relative frequency of join attribute value b_j in relation R_i. By definition, the relative frequency $f_i(x)$ for any $x \in D$ is the fraction of tuples in relation R_i with join attribute value equal to x. Obviously, the product $f_i(x)|R_i|$ is equated to the number of tuples in relation R_i whose the join attribute value is equal to x. Given the relative frequencies of the join attribute values $b_1, b_2, \ldots, b_m$, the join selectivity of $R_1 \bowtie R_2$ is equal to [6]

$$\mu = \sum_{b_j \in D} \prod_{i=1}^{2} f_i(b_j) = \sum_{b_j \in D} f_1(b_j) f_2(b_j) \tag{1}$$

Since $\mu = \frac{|R_1 \bowtie R_2|}{|R_1 \times R_2|}$ and the size of the result set of the cross product $R_1 \times R_2$ is equal to the product $|R_1||R_2|$, the cardinality of the result set associated with the join operation $R_1 \bowtie R_2$ is rendered by the magnitude of the join selectivity. In appendix, a probabilistic meaning to the join selectivity measure is given as well as it is proven an analytical formula about the size of the result set of the chain join, which is a common form of join.

2.1 Homogeneous Input Relations

Firstly, we examine the natural join of two homogeneous relations $R_1(A, B) \bowtie R_2(B, C)$ in the context of the join product skew effect. In the case of the homogeneous relations the distribution of the join attribute values b_i is the same for both input relations R_1 and R_2. That is, there exists a distribution f such that $f_1(b) = f_2(b) = f(b)$ for any $b \in D$. In this setting, the distribution f is skewed when there are join attribute values $b_i, b_j \in D$ such that $f(b_i) \gg f(b_j)$.

The join attribute values with the same relative frequency f_k defines the *frequency class* $C_k = \{b \in D \mid f(b) = f_k\}$.

Thus, the domain D of the join attribute B is disjointly separated into classes of different frequencies. This separation can be represented with a two level tree, called *frequency tree*. The nodes of the first level correspond to classes of different frequencies. The k^{th} node of the first level is labeled with C_k. The descendant leaves of the labeled node C_k correspond to the join attributes belonging to class C_k. Each leaf is labeled with the value of one of the join attributes of the class corresponding to the parent node. The following picture depicts the

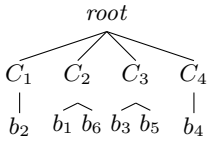

Fig. 1. The frequency tree for $R_1 \bowtie R_2$.

structure of a simple frequency tree for join operation $R_1 \bowtie R_2$ assuming that $D = \{b_1, \ldots, b_6\}$ is separated into four frequency classes $C_1, \ldots, C_4$.

The number of produced joined tuples for a given class C_k is equal to $|C_k| f_k^2 |R_1||R_2|$ since $f_k|R_1|$ tuples of relation R_1 matches with $f_k|R_2|$ tuples of relation R_2 on any join attribute value $b \in C_k$. Let N be the number of the database processors participating in the computation of the join operation. Since only the join product skew effect is considered, the workload associated with each node is determined by the size of the partial result set that is computed locally. In order the workload of the join operation to be evenly apportioned on the N database processors, each node should produce approximately $\left(\frac{\sum_{k=1}^{K} |C_k| f_k^2}{N}\right)|R_1||R_2|$ number of joined tuples, where K denotes the number of frequency classes. In terms of the frequency classes, this is equivalent to an appropriate assignment of either entire or subset of frequency class(es) to each database processors in order to achieve the nearly even distribution of the workload. This assignment can be represented by the selection of some internal nodes and leaves in the frequency tree. By construction, the selection of an internal node in the frequency tree amounts to the exclusive assignment of the corresponding frequency class to some database processor. Thus, this database processor will join tuples from the relations R_1 and R_2 whose join attribute value belongs to the selected class. Finally, to guarantee the integrity of the final result set, the sequence of selections must span all the leaves of the frequency tree.

2.2 Heterogeneous Input Relations

We extend the previous analysis in the case of heterogenous input relations. The join attribute values are distributed to the input relations $R_1(A, B)$ and $R_2(B, C)$ according to the data distributions f_1 and f_2, respectively. In general, it holds that the relative frequencies of any join attribute value $b \in D$ are different in the relations R_1 and R_2, i.e., $f_1(b) \neq f_2(b)$ for any $b \in D$. The above are depicted in table 1.

The number of joined tuples corresponding to the join attribute value $b \in D$ is rendered by the product $f_1(b) f_2(b)$. Thus, the join product skew happens when $f_1(b_i) f_2(b_i) \gg f_2(b_j) f_2(b_j)$ for some $b_i, b_j \in D$. This means that the workload of the join process for the database processor, to which the tuples with join attribute value equal to b_i have been shipped at the redistribution phase, will be disproportional compared with the respective workload of another database processor. Similarly to section 2.1, the classes $C_k = \{b \in D \mid f_1(b) f_2(b) = f_k\}$ disjointly partition the join attribute values.

Join Attribute Values	R_1	R_2
b_1	$f_1(b_1)$	$f_2(b_1)$
...	...	...
b_m	$f_1(b_m)$	$f_2(b_m)$

Table 1. Relative frequencies of the join attribute values.

Alternatively, it is possible the definition of classes of ranges of frequencies according to the schema $C_k = \{b \in D \mid f_{k-1} \leq f_1(x)f_2(x) < f_k\}$ (range partitioning in the frequency level).

The "primary-key-to-foreign-key" join consists a special case of heterogeneity where in one of the two relation, say R_1, two different tuples always have different values in the attribute B. This attribute is called primary key and its each value $b \in D$ uniquely identifies a tuple in relation R_1. As to relation R_2, attribute B, called foreign key, matches the primary key of the referenced relation R_1. In this setting, which is very common in practice, we have that $f_1(b_i) = \frac{1}{m}$ for any $b_i \in D$, and in general $f_2(b_i) \neq \frac{1}{m}$ with $f_2(b_i) > 0$. The join product skew happens when $f_2(b_i) \gg f_2(b_j)$ for some $b_i, b_j \in D$, since $f_1(b_i) = f_1(b_j)$. Thus, the separation of the join attribute values into disjoint frequency classes can be defined with respect to the data distribution f_2, i.e., $C_k = \{x \in D \mid f_2(x) = f_k\}$.

3 Algorithm

In the continue we propose an algorithm that handles join product skewness. The algorithm deals with the simple case of the binary join operation between two relations $R(A, B) \bowtie S(B, C)$, in which the join predicate is B. For the explanation of the algorithm we use the following notation. Let D be the domain of values of the join attribute then $D = \{b_1, b_2, ..., b_m\}$ where m is the number of the discrete join attribute values.

$|R_b i|$ is the number of tuples of the relation R with join attribute value equal to b_i for every $i = \{1, 2, ..., m\}$.

$|S_b i|$ is the number of tuples of the relation S with join attribute value equal to b_i for every $i = \{1, 2, ..., m\}$.

The algorithm considers $|R_b i|$ and $|S_b i|$ given for every value of i. The number of computations that should take place in order to commit the join operation depends on the number of tuples from both relations that have common join attribute value ($R.B = S.B$) and it is equal to $|R \bowtie S|$. In other words the total process cost for the join operation TPC is given by the sum of products of the tuples in both relations that have equal join attribute values and it is expressed by the equation

$TPC = \sum_{b_i \in D} |R_b i| * |S_b i|$.
We notate by n the given number of processors. If we divide the TPC by n we get the number of computations that should each processor commits in order to avoid delays in one or more processors.
The symbol *pwl* stands for the work load that corresponds to each processor.
The quotient of the division of the product of $|R_b i|$ * $|S_b i|$ by *pwl* gives the number of processors needed to handle each distinct join attribute value. If this number for which we use the symbol $vwl_{b_a j}$ (work load for the join attribute value $b_a j$) exceeds value 1 the algorithm considers the join attribute value skewed and decides to send this join to more than one processors. The exact number is defined by the quotient. Let $SK = \{b_a 1, b_a 2, b_a 3, ..., b_a l\}$ be the skewed values. For each one of them the algorithm's steps are the following. For the value $b_a 1$ the number of processors needed is $vwl_{b_a 1}$. The algorithm uses the first $vwl_{b_a 1}$ processors for the join attribute value $b_a 1$. For this step we need to know which of the two relations has the greater number of tuples. If $|R_{b_a 1}| > |S_{b_a 1}|$ the tuples of the relation R are redistributed to the first $vwl_{b_a 1}$ processors while all the tuples from the second relation are send to all $vwl_{b_a 1}$ processors. In order to decide which tuple of the relation R goes to which processor the algorithm uses a hash function on a different attribute from the join attribute. So at this step all processors apply the hash function to the tuples from relation R whose join attribute value equals to $b_a 1$ and send them to the appropriate processor. In the opposite case where $|R_{b_a 1}| < |S_{b_a 1}|$ all the tuples from the the first relation with join attribute value equal to $b_a 1$ go to all the processors and the tuples of the second relation are distributed to all $vwl_{b_a 1}$ processors according to the hash function. The same procedure takes place for the rest skewed values. The remaining tuples are redistributed to the the rest processors according to the hash function which is applied to the join attribute value.

4 Related Work

The achievement of load balancing in the presence of redistribution and join product skew is related to the development of static and dynamic algorithms. In static algorithms it is assumed that adequate information on skewed data is known before the application of the algorithm. [1], [3], [5], [15], [14], [16], [18] expose static algorithms. On the contrary, data skew effect is detected and encountered dynamically at run time [2], [7], [8], [9], [19].

[8], [9] meet the challenge of redistribution skew in the context of a dynamic approach. According to the bucket-spreading parallel hash join algorithm [9] and its variant tuple interleaving hash join algorithm [8], redistribution skew is alleviated by guaranteeing that the database processors get approximately the same number of tuple for the join phase. However, to achieve this the relations are redistributed twice.

[2], [19] address the issue of the join product skew following a dynamic approach. A dynamic parallel join algorithm that employs a two-phase scheduling procedure is proposed in [19]. The authors of [2] present an hybrid frequency-

Algorithm HJPS (Handling Join Product Skew *)*
Input: t_{r_i} tuples of relations R and t_{r_j} tuples of relations S, number of processors n.
Output: correspondence of tuple to processor
Consider the join attribute value is the set:
$D = \{b_1, b_2, ..., b_m\}$
(* compute all frequencies for every join attribute value inD *)
for $i := 1$ **to** m **do**
    calculate $|R_b i|$, $|S_b i|$;
$TPC = \sum_{b_i \in D} |R_b i| * |S_b i|$ (*TCP the total process cost*)
$pwl = TPC/n$
(*pwl the process cost of each processor*)
$vwl_{b_i} = |R_b i| * |S_b i|$;
(*vwl_{b_i} the process cost for each join attribute value b_i)
$vn_{b_i} = vwl_{b_i}/pwl$;
(*pn_{b_i} ideal number of processors for the join attribute value b_i*)
if ($pn_{b_i} >= 2$) consider b_i a skewed value;
Let $SK = \{b_{a_1}, b_{a_2}, b_{a_3}, ..., b_{a_l}\}$ be the set of skewed values
for $i := a_1$ **to** a_m **do**
    if ($|R_b i| > |S_b i|$)
        distribute every t_{r_i} to the next vn_{b_i} processors;
        send every t_{s_i} to the next vn_{b_i} processors;
    else
        distribute every t_{s_i} to the next vn_{b_i} processors;
        send every t_{r_i} to the next vn_{b_i} processors;
assign rest tuples from both relations to the rest processors;

(*for distribution use a hash function to another attribute*)
(*the algorithm uses a hash function to the join attribute*)

Fig. 2. Handling Join Product Skew Algorithm

adaptive algorithm dynamically combines histogram-based balancing with standard hashing methods. The main idea is that the processing of each sub-relation, stored in a processor, depends on the join attribute value frequencies which are determined by its volume and the hashing distribution.

[1] , [5], [15], [14] deal with the redistribution skew effect assuming perfect or nearly perfect information of the join attribute distribution. The algorithm in [15] splits the hash phase into two phases and adds a scheduling phase between these two. During the scheduling phase, an heuristic optimization algorithm balance the load across the multiple processors. [14] follows the same approach for sort-merge join by adding a scheduling phase. During this phase the balance of the workload is achieved using the output of the sort phase. Subset replication and range partitioning techniques, presented in [5], computes a split vector of join attribute values given a set of collected samples of the relations to be joined. The split vector is used to partition the skewed relation into a number of ranges that is equal to the processors. In PRPD algorithm [18], the tuples of each sub-relation of R_1 with skewed join attribute values occurring in R_1 are kept locally in the database processor, while the set of tuples that have skewed values happening in R_2 are broadcast to all the database processor. The remaining tuples of sub-relation are hash redistributed. The remaining tuples of each sub-relation of R_2 are treated in the respective way. The algorithm captures efficiently the case where some values are skewed in both relations.

Virtual processor partitioning [5] is designed to deal with the presence of the product skew statically. Using the notion of the splitting values stored in a split vector, multiple range partitions instead of one are assigned to each processor. Authors in [1] assign a work weight function to each join attribute value in order to generate partitions of nearly equal weight.

Finally, OJSO algorithm [17] handles data skew effect in an outer join, a variant of the equi-join operation.

5 Conclusion and Future Work

We address the problem of join product skew in the context of the PDBMS. In our analysis, the apriori knowledge of the distribution of the join attribute values has been taken for granted. We concentrated on the case of partitioned parallelism, according to which the join operator to be parallelized is split into many independent operators each working on a part of data. We introduce the notion of frequency classes and we examined its application in the general cases of homogeneous and heterogeneous input relations. Furthermore, an algorithmic framework called HJPS is proposed to handle the join product skew. The proposed algorithm identifies the skew elements and assigns a specific number of processors to each of them. Given a skewed join attribute value, the number of dedicated processors is rendered by the process cost for computing the join for this attribute value, and by the workload that a processor can afford.

Much work needs to be done for our work to mature. We are in the process of testing our algorithm against real databases, and a benchmark analysis should

be conducted as well. In addition, we are looking at generalizing our framework analysis with frequency classes at multiple joins. The study of the chain join in appendix section assuming non-correlated join attribute values is only a special case.

References

1. K. Alsabti and S. Ranka. Skew-insensitive parallel algorithms for relational join. In *HIPC '98: Proceedings of the Fifth International Conference on High Performance Computing*, page 367, Washington, DC, USA, 1998. IEEE Computer Society.
2. Mostafa Bamha and Gaétan Hains. Frequency-adaptive join for shared nothing machines. pages 227–241, 2001.
3. Hasanat M. Dewan, Mauricio A. Hernández, Kui W. Mok, and Salvatore J. Stolfo. Predictive dynamic load balancing of parallel hash-joins over heterogeneous processors in the presence of data skew. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, September 28-30, 1994*, pages 40–49. IEEE Computer Society, 1994.
4. David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.
5. David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In Li-Yan Yuan, editor, *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*, pages 27–40. Morgan Kaufmann, 1992.
6. Peter J. Haas, Jeffrey F. Naughton, and Arun N. Swami. On the relative cost of sampling for join selectivity estimation. In *PODS '94: Proceedings of the thirteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 14–24, New York, NY, USA, 1994. ACM.
7. Lilian Harada and Masaru Kitsuregawa. Dynamic join product skew handling for hash-joins in shared-nothing database systems. In Tok Wang Ling and Yoshifumi Masunaga, editors, *Database Systems for Advanced Applications '95, Proceedings of the 4th International Conference on Database Systems for Advanced Applications (DASFAA), Singapore, April 11-13, 1995*, volume 5 of *Advanced Database Research and Development Series*, pages 246–255. World Scientific, 1995.
8. Kien A. Hua and Chiang Lee. Handling data skew in multiprocessor database computers using partition tuning. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 525–535. Morgan Kaufmann, 1991.
9. Masaru Kitsuregawa and Yasushi Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (sdc). In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 210–221. Morgan Kaufmann, 1990.
10. M. Seetha Lakshmi and Philip S. Yu. Effectiveness of parallel joins. *IEEE Trans. Knowl. Data Eng.*, 2(4):410–424, 1990.
11. Manish Mehta and David J. DeWitt. Data placement in shared-nothing parallel database systems. *VLDB J.*, 6(1):53–72, 1997.
12. Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

13. Christopher B. Walton, Alfred G. Dale, and Roy M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 537–548. Morgan Kaufmann, 1991.
14. Joel L. Wolf, Daniel M. Dias, and Philip S. Yu. An effective algorithm for parallelizing sort merge in the presence of data skew. In *DPDS*, pages 103–115, 1990.
15. Joel L. Wolf, Daniel M. Dias, Philip S. Yu, and John Turek. An effective algorithm for parallelizing hash joins in the presence of data skew. In *Proceedings of the Seventh International Conference on Data Engineering, April 8-12, 1991, Kobe, Japan*, pages 200–209. IEEE Computer Society, 1991.
16. Joel L. Wolf, Daniel M. Dias, Philip S. Yu, and John Turek. New algorithms for parallelizing relational database joins in the presence of data skew. *IEEE Trans. Knowl. Data Eng.*, 6(6):990–997, 1994.
17. Yu Xu and Pekka Kostamaa. Efficient outer join data skew handling in parallel dbms. *PVLDB*, 2(2):1390–1396, 2009.
18. Yu Xu, Pekka Kostamaa, Xin Zhou, and Liang Chen. Handling data skew in parallel joins in shared-nothing systems. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1043–1052, New York, NY, USA, 2008. ACM.
19. Z. Xiaofang and M.E. Orlowska. Handling data skew in parallel hash join computation using two-phase scheduling. In *Algorithms and Architectures for Parallel Processing*, pages 527 – 536. IEEE Computer Society, 1995.

Appendix

We consider the case of the chain join (which is one of the most common form of join) in terms of finding an analytical formula about the size of the result set. A chain join is a join of the form

$$R = R_1(A_0, A_1) \bowtie R_2(A_1, A_2) \bowtie \ldots \bowtie R_k(A_{k-1}, A_k)$$

where each relation R_i joins with the following one R_{i+1} on the single join attribute A_i. It is possible that there are additional attributes in the schema of R_i's. However, for convenience we omit to include attributes that do not participate in the join process. In our analysis, we denote by $\mu_{i,i+1}$ the selectivity of the join operation $R_i(A_{i-1}, A_i) \bowtie R_{i+1}(A_i, A_{i+1})$ on attribute A_i, where $1 \leq i \leq k-1$. The join selectivity $\mu_{i,i+1}$ is considered as the probability of the event that two randomly picked tuples belonging to relations R_i and R_{i+1} respectively join on the same join attribute value. In the following lemma, it will be proved that the selectivity of the chain join is equal to the product of the selectivities of the constituent binary join operations under the condition that there is no dependence between the values of the join attributes A_i.

Lemma 51 *Given that the values of the join attributes A_i in a chain join of k relations are independent of each other, the overall join selectivity of the chain join, denoted by μ, is equal to the product of the selectivities of the constituent binary join operations, i.e., $\mu = \prod_{i=1}^{k} \mu_{i,i+1}$.*

Proof. We define a pair of random variables $(\mathsf{X}_i, \mathsf{Y}_i)$ for every relation R_i, where $i = 2, \ldots, k-1$. Specifically, the random variable X_i corresponds to the join attribute $R_i.A_i$ and it is defined as the function $\mathsf{X}_i(t) : \Omega_i \rightarrow \mathbb{N}_{\mathsf{X}_i}$, where Ω_i is the set of the tuples in the relation R_i. $\mathbb{N}_{\mathsf{X}_i}$ stands for the set $\{0, 1, \ldots, |D_{A_i}| - 1\}$, where D_{A_i} is the domain of the join attribute A_i. In other words, $\mathbb{N}_{\mathrm{X}_i}$ defines an enumeration of the values of the join attribute A_i, in such a way that there is a one-to-one correspondence between the values of the set D_{A_i} and $\mathbb{N}_{\mathsf{X}_i}$. Similarly, the random variable $\mathsf{Y}_i(t) : \Omega_i \rightarrow \mathbb{N}_{\mathsf{Y}_i}$, where $\mathbb{N}_{\mathsf{Y}_i}$ represents the set $\{0, 1, \ldots, |D_{A_{i+1}}| - 1\}$, corresponds to the join attribute A_{i+1}. As for the relations R_1 and R_k, only the random variables Y_1 and X_k are defined, since the attributes $R_1.A_0$ and $R_k.A_k$ do not participate in the join process.

Let $\mathcal{R}$ denote the event of the join process. Then we have that

$$p(\mathcal{R}) = p\big(\mathsf{Y}_1 = \mathsf{X}_2 \wedge \mathsf{Y}_2 = \mathsf{X}_3 \wedge \ldots \wedge \mathsf{Y}_{k-1} = \mathsf{X}_k\big)$$

By assumption, the random variables are independent of each other. Thus, it is valid to say that $p(\mathcal{R}) = \prod_{i=1}^{k-1} p(\mathsf{Y}_i = \mathsf{X}_{i+1})$. Moreover, $p(\mathsf{Y}_i = \mathsf{X}_{i+1})$ represents the probability of the event that two randomly picked tuples from relations R_i and R_{i+1} agree on their values of the join attribute A_i. Since it holds that $p(\mathsf{Y}_i = \mathsf{X}_{i+1}) = \mu_{i,i+1}$, the lemma follows.

As a direct consequence of the previous lemma, the cardinality of the result set associated with the chain join of k relations is given by the formula

$$|R| = \big(\prod_{i=1}^{k-1} \mu_{i,i+1}\big) \cdot \prod_{j=1}^{k} |R_j|$$