

An Object-Oriented Framework for the Parallel Join Operation

Sergio Carvalho Alberto Lerner Sérgio Lifschitz
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
Departamento de Informática – Rio de Janeiro, RJ, Brasil
e-mail: {sergio,lerner,lifschit}@inf.puc-rio.br

Abstract

We propose an object-oriented framework for one of the most frequent and costly operations in parallel database systems: the parallel join. The framework independently captures a great variety of parameters, such as different load balancing procedures and different synchronization disciplines. The framework addresses DBMS flexibility, configuration and extensibility issues, via the instantiation of known algorithms and facilities for the introduction of new ones. The framework can also be used to compare algorithms and to determine the execution scenario an algorithm is best suited for. Related algorithms are grouped in families, suggesting a taxonomy.

1. Introduction

Since the introduction of parallel processing architectures, a great deal of effort has been spent on the development of algorithms for relational operators that support intra-operation parallelism – one of the available techniques for DBMS parallelization [1]. Particularly, the join operator gained much attention due to its high cost and extreme importance. As a result, efficient join algorithms have been published [2, 7]. Although quite effective individually, the study of the ensemble of those algorithms raises a number of questions.

First, consider adaptability. As there is no solution suited for every single execution scenario, algorithms should be able to adapt themselves to the scenario being considered. The combination of a number of solutions to address a large range of scenarios is seldom seen.

Second, consider extensibility. Algorithms almost never facilitate the reuse of the mechanisms and techniques they are based on. We need to produce new algorithms based on known techniques or improved techniques based on existing ones.

Finally, consider comparison. As target platform parameters, load-balancing techniques and analytical models differ when distinct algorithms are considered, algorithm comparison is hardly accomplished.

To address these issues, we used object-oriented techniques [3] to design a framework for the parallel join

operation, the PJ framework. Basically, it models a large number of algorithms published in the literature, regardless of their parallelization strategy, their target parallel architecture and their load-balancing techniques. The PJ framework provides support services commonly needed by the algorithms, and promotes the reuse of code, design and domain expertise. In addition, it also facilitates the design of new algorithms.

The remainder of this paper is organized as follows. In Section 2, we introduce the main ideas underlying the PJ framework. In Section 3, we describe its high level design. In section 4, we describe framework dynamics. In Section 5, we discuss related work. Conclusions are found in Section 6.

2. PJ Framework Basics

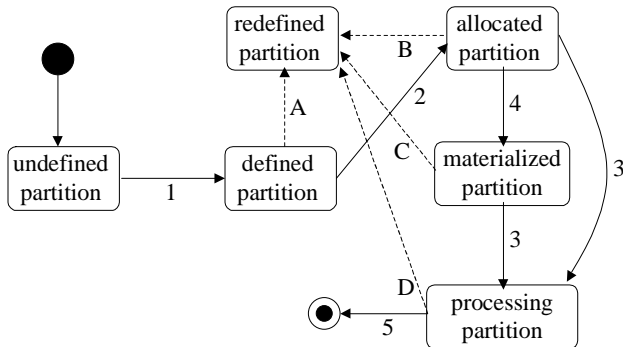
The perception that the vast majority of parallel join algorithms is based on a data parallelism approach (*i.e.*, the division of data in disjoint sets and their further parallel processing) led us to a first step towards the unification of a collection of join algorithms into a single object-oriented framework. In this section we describe the underlying concepts that made this unification possible: partition life-cycles, roles and actors.

2.1. Data Partitions and their States

Data partitions are join independent subsets of tuples of both operand relations. Regardless of the algorithm, throughout the join process, data partitions go through a number of specific states. Initially, every partition is created in an *undefined* state.

Then, a newly created partition must be assigned its share of data, that is, be *defined* or given an identity (*e.g.*, the partition made of tuples of range x ; the partition made of tuples of buckets y and z). A defined partition can then be *allocated* for processing on a processing node (PN). Depending on the target parallel architecture, the partition is or is not physically transferred to its assigned node before (or in the process of) running the so called *local join*. For example, such a materialization is performed in the case of share-nothing architectures. At this point, the allocated partition goes through the join execution, using

a sequential strategy previously defined. The local join runs through completion, and the partition is considered as *processed*. The diagram in Figure 1 represents this partition life-cycle.



1 - portion of data definition	A - overload on partition definition
2 - processing site selection	B - overload on partition allocated
3 - start of local join processing	C - overload on partition moved
4 - physical partition transfer	D - overload on processing partition
5 - end of local processing	

Figure 1 - State transition diagram of the join of one partition

Considering a set of partitions, instead of just a single one, different parallel join algorithms may handle partitions in distinct ways. Algorithms can perform the procedures just described in a partition-by-partition fashion, in loose synchronicity. In this case, partitions may be in different states at any given point in time. On the other hand, all partitions may be forced through the same states at the same time, via a synchronous algorithm based on very well-defined phases.

2.2. Support for Load Balancing

Parallel join algorithms suffer great penalties in the presence of skewed data distributions [8]. A naive data partitioning approach may generate uneven workload distributions among the nodes of the parallel machine. Therefore, the state transition diagram of Figure 1 also carries dashed transitions labeled with letters, to capture uneven data distributions and load imbalances. Note that partition overloads may be detected in different states.

Every approach to regaining workload balance involves the redefinition and the reallocation of a single or a group of partitions.

In the single partition case, we can look at load balancing as if a previously defined partition was found to be *overloaded*, i.e., complete processing would take its assigned PN longer than any other node. To obtain reasonable response times under skewed distributions, this time difference should be kept small.

As a result of overload detection, a partition is divided

and part of it is reallocated. The original partition *unloads some weight* and goes on from the state where it was before the procedure took place. A new partition, born to accommodate the extra weight, may start in the same state as its originating partition or in another state, depending on the load-balancing strategy. Either way, the new partition starts its own life-cycle.

The process just described does not limit itself to a single partition division. It also captures the possibility of splitting an overloaded partition in any number of new ones, enough to level the extra workload across the parallel machine. Moreover, load balancing may also take place for several partitions simultaneously. The newly generated partitions may be assigned the overload of several formerly heavy partitions.

2.3. Roles and Actors

In our approach, a parallel join algorithm is the entity that coordinates the behavior of partitions. Thus, an algorithm is responsible for sending messages to partitions, to cause state transitions. For instance, transition 1 in Figure 1 takes a partition from an *undefined state* to a *defined* state. The message that triggers this transition is the *define-yourself* message. Being purely reactive, partitions cannot define themselves; they need the algorithm to tell them how to do so.

Roles describe the operations that a particular algorithm wants a partition to undertake, thus changing states. In the example, the procedure executed by a partition in response to the *define-yourself* message is specified by the *RelationChopper* role, which splits the relation in independent subsets.

As every role, *chopping* can be accomplished in a number of ways. Three of them particularly stand out: range partitioning, hash partitioning, and fragment-and-replicate.

An *actor* is a strategy used to implement an operation specified by a role. Given the diversity of strategies in which algorithms perform similar operations, a role may have a number of different actors able to play it. In the example, for a partition to define itself, the algorithm defines the actor to be used.

In the next section, we describe how object-oriented techniques were used to combine partition state diagrams and a collection of actors representing roles to build a framework able to incorporate the behavior of a large number of parallel join algorithms.

3. The PJ Framework: Structure

The high-level design of the PJ framework is depicted by the class diagram in Figure 2. The framework consists of three major groups of classes: the *Partition Pool*, the *Role Pool*, and the *Algorithms*, classified into *Families*. The description of each of these groups, to follow, is

greatly facilitated by the extensive use of Design Patterns [4].

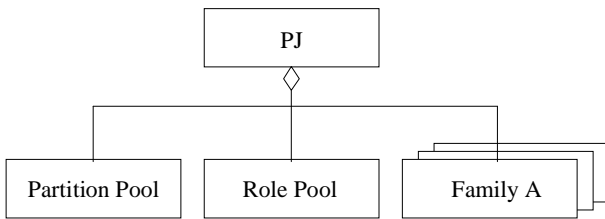


Figure 2 - The PJ Framework high-level structure

The PJ framework is built as a part-of hierarchy, with a Mediator (Singleton) class to facilitate communications and to decouple its components, the class groups just discussed. The mediator also provides an interface for external DBMS contexts where the PJ framework is supposed to be inserted.

3.1. The Partition Pool

The Partition Pool (PP) is responsible for treating all partition issues and encapsulating the life-cycle concept described earlier. It is also modeled as a Singleton class. The PP consists of a collection of Partition objects exclusively accessed via the Pool, and provides facilities for addressing its components in diverse ways.

Each partition, represented by a Partition class object, is responsible for keeping track of its state, as suggested in the State pattern. Partition State classes model the state transition diagram for partitions, redefining valid state operations in its subclasses.

3.2. The Role Pool

The Role Pool (RP) is responsible for capturing the operations that can be performed on partitions. Each role is represented by a Strategy pattern, and is implemented by a variety of Actors, its concrete subclasses.

Each Figure 1 transition has a corresponding Role. Transitions 1 through 4 are represented respectively by *RelationChopper*, *NodeSelector* (assigns defined partitions to nodes), *LocalJoiner* (executes local joins on assigned partitions), and *TupleCollector* (physically move tuples between nodes).

Roles know what to do to make partitions change state. Whenever an algorithm wants a partition to evolve, it sends a message to that partition along with the actor implementing the specific role. The partition interacts with this actor to get the job done.

The additional roles *WorkloadGauge*, *Overload Trigger*, *LoadLeveler*, *PartitionSelector*, and *PartitionMover* are related to distribution and load

balancing. These roles are modeled after Watt and Taylor's work, which divides the load balancing problem in phases [9].

3.3. The Algorithms and their Families

Algorithms use roles and actors to perform parallel joins. Algorithms presenting similar behavior are factored out into Families. Algorithm families use the Template Method pattern. Indeed, it is a common practice in the parallel join literature for a paper to introduce a family of highly similar algorithms (though each directed toward a different execution scenario), as in [5, 2].

A family usually gathers algorithms that share a common structure, but choose distinct actors to play a specific role. For example, in [2], algorithms are proposed based on different choices for the *Relation Chopper* role.

4. The PJ Framework: Behavior

In this section we introduce the main collaboration sequences among PJ framework components. The first sequence refers to the interaction between algorithms, actors, and partitions. In addition, we describe the two sequences involved in load-balancing issues.

4.1. Algorithms = Actors + Partitions

One of the main interactions that takes place in the PJ framework is that of an algorithm sending an execution request to a partition (or a group thereof). Along with the request, the algorithm sends information about the actor to be used. This actor must be able to play the role specified in the operation. The receiving partition then calls the specified actor to perform the operation. We now describe in detail how this interaction takes place inside the framework.

An algorithm starts this sequence by sending the mediator a *partition-do-something* message. The algorithm may or may not wait for the message to be processed, implementing loose or tight synchronicity, respectively, between algorithms and partitions.

The message carries pieces of information that will be used on its way down to the targeted partitions. Upon reception, the mediator checks what type of actor the algorithm specified. It then instantiates a running actor based on the chosen role strategy. The algorithm may have also sent actor setup parameters, so the mediator passes them on to the selected actor.

Subsequently, the mediator forwards the message and the selected actor to the PP, which selects the partitions carried by the message (e.g., all of them, the current one, the next one, those who meet criterion y , etc). The PP is also instructed about which group communication pattern is going to be used between the pool and the partitions (e.g., message broadcast, one-to-one iteratedly, etc).

From this point on, each designated partition interacts with the actor previously instantiated by the mediator. Actually, a partition does not know the specific actor it is interacting with – it only knows the abstract class modeling a Role. The Template Method pattern is used in the Role class hierarchy to specialize behavior in its subclasses. Moreover, the specific role to be called in each state transition is resolved via the Partition class, a State pattern. All a partition has to do is to call the method modeled by the abstract role and pass itself as a parameter. In addition, Proxy classes are used whenever this call crosses processor boundaries.

An example interaction Take the split phase of the ABJ algorithm [5], for instance. It partitions both relations in a large number of buckets and allocates them round-robin to processors, thus carrying the corresponding tuple movement.

ABJ's split phase under the PJ Framework uses a Hash-Based *RelationChopper*, a Round-Robin *Node Selector*, and a Hashed-Scan *TupleCollector*.

After the definition of partitions the mediator initializes the Hash-Based actor with the number of partitions and the data distribution histogram (e.g., normal distribution). On behalf of the algorithm, all partitions, in a one-by-one fashion, are told to call the role for the current transition – the previously instantiated Hash-Based *RelationChopper* – to have their ranges defined. Upon each call, the Hash-Based *RelationChopper* assigns a new set of hash buckets to the calling partition, taking it to the next state.

The ABJ and the mediator then proceed in similar interactions to assign a PN to the recently defined partition, to move the partition to the assigned node, and finally to collect each partition's tuples.

4.2. Load Monitoring

The handling of the data skew phenomenon is one of the most sophisticated features of parallel join algorithms, and the previous sequence is not able to capture a good variety of such strategies. We now focus on an additional interaction that, combined with the one to be shown in Subsection 4.3, fully supports the load-balancing techniques presented in the most significant algorithms.

This interaction is mainly provided to support the framework's workload-monitoring capability. Monitoring the workload throughout algorithm execution is the first step towards the correction of load imbalances. Algorithms define distinct phases and strategies to undertake monitoring. Whenever monitoring yields an *overload* message, the algorithm should perform its load balancing strategy to distribute the extra load.

This monitoring capability configures another framework *hot-spot*, which can be considered in two orthogonal aspects: how to measure a partition's load, and when to do it. The role *WorkloadGauge*, responsible for

both aspects, models a number of implementation actors, each with a specific combination of how and when to measure a partition's load.

As for how measurements occur, there is no consensus metric in parallel join algorithms. Metrics mentioned in the literature include tuple cardinality, processing time, and node load. Moreover, tuple cardinality unfolds in actual partition cardinality, actual result cardinality (as the one generated by local joins), and predicted result cardinality. Similarly, processing time may be accounted for as estimated finishing time, actual processing time, or actual time unit per result unit. Although not extensive, this list of alternatives shows how diversely one can measure workload on a parallel join algorithm.

The measurement of a partition's load may occur on distinct occasions. The initiative of updating a partition load may come from the algorithm or from the *WorkloadGauge* actor. In the first case, the algorithm simply uses the sequence of Subsection 4.1 to have the partition interact with the actor to calculate its load. In the latter case, the same sequence is used but only to subscribe the actor as a partition Observer. Using this pattern, the Subject partition is instructed to notify the Observer *WorkloadGauge* actor upon certain events, such as time slices and tuple cardinality thresholds.

Once computed, the measured workload can be confronted either with a previously expected value or with workloads held by other partitions. The *OverloadTrigger* role is responsible for making measured workload comparisons and deciding whether an *imbalance* message should be sent to the running algorithm (via the mediator). The interaction between an *OverloadTrigger* actor and the partitions is similar to that of a *WorkloadGauge* and a partition.

An example interaction Again we consider the ABJ algorithm to illustrate how load monitoring takes place. After the split phase, ABJ performs a partition tuning phase, deciding whether buckets should be reallocated in order to level the workload across processors, taking bucket cardinality as the metric.

Under the PJ Framework, in that stage, the ABJ algorithm asks the mediator to start up a Simple Bucket Cardinality *WorkloadGauge* actor and a Simple Mean Cardinality *OverloadTrigger* actor (the "simple" stands for a once-used actor). The algorithm then instructs the mediator to proceed with a load-checking interaction. As a result, the Simple Mean Cardinality actor may have sent an *imbalance* message, should excess buckets be identified.

By allowing the *imbalance* message to happen also on latter algorithm stages (i.e., by using more sophisticated actors for both the *WorkloadGauge* and the *OverloadTrigger* roles), the ABJ algorithm may gain dynamic load balance behavior [7]. Actually, ABJ suffers JPS skew effects due to its inability to balance the load on later execution stages. By adding dynamic load-balancing

strategies already validated by other algorithms, an ABJ variant under the PJ framework could be made resistant to JPS.

4.3. Load Balancing

Upon receipt of an *imbalance* message, an algorithm built under the PJ framework may engage in a load-balancing stage. The load-balancing interaction starts whenever an algorithm picks actors for the *LoadLeveler* and *PartitionSelector* roles. The *LoadLeveler* role is responsible for determining the ideal work transfer vector necessary to balance execution at that point in time; and the *PartitionSelector* role, for choosing which partitions should be transferred so to best fulfill the vector [9].

The algorithm can then start the selected balancing operation by sending a *load-balance* message to the mediator, which immediately forwards the message to the PP. The PP gathers workloads per processing node and, with the help of *LoadLeveler* and *PartitionSelector* actors, comes to a new partition set configuration. The new configuration is enforced by the PP via the partitions' ability to divide and combine themselves – eventually using a *PartitionMover* actor to physically transfer tuples across nodes.

An example interaction In the ABJ algorithm example, once the existence of excess buckets is determined during the partition tuning phase, the algorithm redistributes the extra load among underloaded processors using a best-fit strategy.

Under the PJ Framework, a Tuple Cardinality *Load Leveler* actor is responsible for determining the amount of tuples to be transferred in order to regain balance across processing nodes. Then, a Shared-Nothing aware *PartitionSelector* determines the most appropriate buckets to be relocated. The PP carries out the resulting partition.

5. Related Work

Lu and Tan developed a framework modeling the relational equi-join operation. In [7], they also considered the partition-oriented approach for parallel join modeling. Lu and Tan's work differs from ours mainly in that they are solely oriented towards a taxonomy of techniques, whereas we present an object-oriented framework.

Watts and Taylor [9] model the general dynamic load balancing problem not using an object-oriented framework. Most of the PJ framework's load balancing capabilities are greatly influenced by their work, including the five-phase decomposition of the load balancing process. They also consider each phase as a hot-spot.

6. Conclusion

This paper introduces the PJ framework, an object-

oriented framework for the parallel join operation in database systems. The framework is based on the perception that parallel join algorithms are inherently data parallel applications. We shortly discuss its underlying ideas and its design, and give an example of its behavior.

The PJ framework addresses important issues in the following ways:

- Configuration – comes in two distinct flavors. First, a DBMS implementor can count on a number of distinct techniques to accomplish the same assignment. Second, a DBMS optimizer can use the PJ's facilities to implement a number of different join solutions in a single operator.
- Extensibility – achieved via abstract operations that allow DBMS implementors to attach additional techniques to the ones already provided.
- Algorithm Comparison – based on the PJ's ability to instantiate distinct parallel join strategies. By being able to not only instantiate them but also, as is the case for a number of solutions, share design decisions, the PJ framework can be used as an algorithm test-bed. Metrics and comparison methods have been specifically discussed in [6].

References

- [1] D. DeWitt and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems". *Communications of the ACM*, 1992, 35(6), pp. 85-98.
- [2] D.J. DeWitt, J.F. Naughton, D.A. Schneider, and S. Seshadri, "Practical Skew Handling in Parallel Joins", *Procs. 18th Intl Conf. on VLDB*, Vancouver, Canada, 1992, pp. 27-40.
- [3] M.E. Fayad and D.C. Schmidt, "Object-Oriented Application Frameworks", *Communications of the ACM*, 1997, 40(10), pp. 32-38.
- [4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994.
- [5] K.A. Hua and C. Lee, "Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning", *Procs. 17th Intl Conf. on VLDB*, Barcelona, Spain, 1991, pp. 525-535.
- [6] A. Lerner, S. Lifschitz, and M.V. Poggi, "An Online Approach for Parallel Join Algorithms Analysis", Technical Report no. 01/99, Depto de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 1999.
- [7] H. Lu and K-L. Tan, "Load-Balanced Join Processing in Shared-Nothing Systems", *Journal of Parallel and Distributed Computing*, 1994, 23, pp. 382-398.
- [8] C.B. Walton, A.G. Dale and R.M. Jenevein, "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins", *Procs. 17th Intl Conf. on VLDB*, Barcelona, Spain, 1991, pp. 537-548.
- [9] J. Watts and S. Taylor, "A Practical Approach to Dynamic Load Balancing", *IEEE Transactions on Parallel and Distributed Systems*, 1998, 9(3), pp. 235-248.