Data-Parallel Computing Meets STRIPS

Erez Karpas and **Tomer Sagi** and **Carmel Domshlak** and **Avigdor Gal** and **Avi Mendelson**
{karpase@tx, stomers7@tx, dcarmel@ie, avigal@ie, avi.mendelson@tce} .technion.ac.il
Technion — Israel Institute of Technology
Technion City
Haifa 32000, Israel

Moshe Tennenholtz
moshet@microsoft.com
Microsoft Research & Technion
Microsoft — Herzliya R&D Center
13 Shenkar St.
Gav-Yam, Building No. 5
Herzliya 46275, Israel [*]

Abstract

The increased demand for distributed computations on big data has led to solutions such as SCOPE, DryadLINQ, Pig, and Hive, which allow the user to specify queries in an SQL-like language, enriched with sets of user-defined operators. The lack of exact semantics for user-defined operators interferes with the query optimization process, thus putting the burden of suggesting, at least partial, query plans on the user. In an attempt to ease this burden, we propose a formal model that allows for data-parallel program synthesis (DPPS) in a semantically well-defined manner. We show that this model generalizes existing frameworks for data-parallel computation, while providing the flexibility of query plan generation that is currently absent from these frameworks. In particular, we show how existing, off-the-shelf, AI planning tools can be used for solving DPPS tasks.

Motivation

In the classical approach to data processing, the user of a database management system (DBMS) specifies her data processing goal as a *declarative query*, and the DBMS automatically generates a *query plan*, i.e., a data processing program that computes the desired goal (Ullman 1988). The final query plan is typically constructed by first generating *some* query plan, and then optimizing it locally by applying a set of predefined query-plan rewriting rules (see e.g., Ambite and Knoblock 2001). This high-level workflow is based on the implicit assumption that query plans consist mostly of built-in operators (for example, relational algebra operators). This assumption is needed because the system cannot proactively select user-defined operators for performing intermediate computations, and query plans cannot be optimized around these user-defined operators.

With the rapid growth of both public and enterprise data repositories, the area of data processing faces conceptual changes in the way the data is perceived, analyzed, and stored. Classical DBMSs and their various extensions are still a cornerstone of data processing, yet large scale and operator-rich computations involving huge amounts of data are becoming more and more common. This, along with the physical limitations on the computing power and storage that a single machine can provide, has led to an increased interest in computation on highly distributed computing infrastructure, often referred to as "data-parallel computing". While parallel and distributed database systems (Valduriez 1993; Ozsu and Valduriez 2007) can address the scalability issues, they, like traditional DMBSs, lack flexibility in exploiting general user-defined operators.

To address this need, several systems for data-parallel computing have been developed; the most well-known of these are probably Map/Reduce (Dean and Ghemawat 2008) and its open-source implementation Hadoop, and Dryad (Isard et al. 2007). These systems are based on low-level query plan programming, which is both error-prone and inhibits mass adoption. However, they allowed for development of higher-level systems, such as SCOPE (Chaiken et al. 2008), DryadLINQ (Isard and Yu 2009), Pig (Olston et al. 2008b), and Hive (Thusoo et al. 2009; 2010), which support combining programming with queries in SQL-like languages. The high-level design of these systems is driven by two primary requirements: operability with arbitrary rich sets of user-defined operators for performing intermediate steps of query plans, and optimization of query plans that takes into account the distributed nature of the underlying computing/storage infrastructure.

Unfortunately, supporting unconstrained usage of user-defined operators comes at the expense of moving away from the declarative approach to data processing. First, if the user formulates a query which departs from the built-in operators, then she must provide the system with a base query plan. Second, to allow for further optimizations of the query plan, the user must explicitly instruct the system on what kind of optimizations can be performed around the non-standard operators.

[*]The last four authors appear in alphabetical order.

Both these requirements take the user away from the goal-driven, declarative data processing paradigm, and currently there is no agreement on what is the right balance between expressivity of the system and the burden to be put on the user: Some systems restrict user-defined operators to be of certain allowed types (Cohen 2006; Chaiken et al. 2008; Thusoo et al. 2010), and others do not optimize user-specified query plans (Olston et al. 2008b). The reason that general user-defined operators pose such a challenge is that the system is uncertain about "what they need" and/or "what their effects are". While the semantics of the built-in operators is "hard-coded" in the design of the data-parallel computing systems, user-defined operators typically come with no semantics attached to them. Although some systems attempt to perform automated analysis of the user's code (Isard and Yu 2009; Cafarella and Ré 2010), these analyses are (and must be) limited, and can not be used to "reverse engineer" the semantics of a given user-defined function.

For example, say an analyst wants two histograms of the users of a large social network, one by age and one by relationship status. The optimal query plan for that request will consist of a single table scan and two counts, but allowing the system to come up with such a plan is tricky. In an SQL-style query language, this request will have to be formulated as two queries, both expressed as aggregations over the users table, and planning for these queries in separation will result in a sub-optimal plan with two scans of the same table. However, suppose that somebody already implemented a "double aggregation" operator, which performs two aggregations over the same table, while only scanning it once. Although that operator can, in principle, be used for devising an optimal query plan, the system does not know what this operator can be used for and how, since this operator is user-defined. Hence, our analyst either settles for a sub-optimal query plan, or, if she is somehow aware of this "double aggregation" operator, then she can provide the system with a base query plan that uses this operator. While this is a simplistic example, it is meant for illustration purposes only; more complex example, e.g. planning for image analysis (Chien et al. 1999), can also can be formulated in terms of our proposed model.

The dilemma above is not new, and it shows up each time a declarative interface to data processing is challenged by the need to expand the expressivity of the system. In particular, the interplay between the usage complexity of the operator description languages, their expressivity, and the computational complexity of automatically composing the operators to achieve a declaratively specified goal is precisely the focus of AI planning research. The question that initiated our investigation was precisely that: What level of expressivity is expected these days from the realm of data-parallel computing, and how/whether AI planning technology can be leveraged to achieve this expressivity while reviving declarative data processing.

To study this question, we suggest a formal model of data-parallel program synthesis, DPPS, that generalizes the specific systems developed in the area. The expressivity of DPPS goes beyond the expressivity of relational algebra extended with aggregate functions, allowing for both taking into account the distributed nature of data-parallel computation, as well as incorporating arbitrary user-defined operators of the form supported by the current data-parallel computing systems. While this level of expressivity unavoidably makes program synthesis and optimization in DPPS computationally hard in the worst case, the syntax and semantics of the DPPS model bear close similarity to the standard action scheme models adopted in the field of AI planning. Exploiting that, we show how DPPS tasks can be compiled into STRIPS planning tasks, and thus solved by off-the-shelf AI planning tools, even if not all possible artifacts of the DPPS operators known to the system can be enumerated in polynomial time.

Model

Our formalism of data-parallel program synthesis is tailored to tracking computation-specific *data chunks*. Each such data chunk represents some information which can be either generated or required as input by a computation primitive. For example, a data chunk can represent all records of males between the ages of 18–49, or the average salary of all males between the ages of 18–49, etc. Note that the actual value of the average salary, does not need to be known in advance; the fact that it is *possible* to compute this average given the input records suffices.

Formally, a data-parallel program synthesis (DPPS) task consists of:

- D — a set of possible data chunks. Each data chunk d is associated with the amount σ_d of memory it requires (given, for example, in MB). D may be given explicitly or described implicitly, in which case it could even be infinite.
- N — a finite set of computing units, or processors. Each processor n is associated with the maximum total size κ_n of the data chunks it can hold and access efficiently.
- A — a set of possible computation primitives. Each such primitive is a triplet $a = \langle \bar{I}, \bar{O}, C \rangle$, where $\bar{I} \subseteq D$ is the required input, $\bar{O} \subseteq D$ is the produced output, and $C : N \rightarrow \mathbb{R}^{0+}$ is a function describing the cost of the computation on each processor. Similarly to the possible data chunks, the set of computation primitives can also be given explicitly or implicitly, and in the latter case, the set A could be infinite.
- $T : N \times D \times N \rightarrow \mathbb{R}^{0+}$ — the data transmission cost function. $T(n_1, d, n_2)$ is the cost of transmitting data chunk d from processor n_1 to processor n_2.
- s_0 — the initial state of the computation.
- G — the goal of the computation.

A state of the computation describes which chunks of data each processor currently holds. Formally, a state $s : N \rightarrow 2^D$ maps each processor to the set of data chunks it holds. For the sake of convenience, we define the free memory capacity of processor n at state s by $f(n, s) := \kappa_n - \sum_{d \in s(n)} \sigma_d$. The goal is also a function $G : N \rightarrow 2^D$ that maps processors to data chunk sets, describing which chunks of data should be stored at each processor at the end of the computation. A state s satisfies the goal G iff each processor holds all the data chunks specified by the goal, that is, $G(n) \subseteq s(n)$ for all $n \in N$.

The semantics of a DPPS task is as follows. A computation primitive $a = \langle \bar{I}, \bar{O}, C \rangle$ can run on processor $n \in N$ at state s iff n holds all the input data chunks and has enough free memory to hold the output, that is, if $\bar{I} \subseteq s(n)$ and $\sum_{d \in \bar{O}} \sigma_d \leq f(n, s)$. Performing a then generates the output data chunks at n, while incurring a cost of $C(n)$. That is, if $s[\![a_n]\!]$ is the state resulting from applying a at processor n in state s, then $s[\![a_n]\!](n) = s(n) \cup \bar{O}$, *ceteris paribus*. Considering the connectivity, a processor n_1 holding data chunk d can transmit it to processor n_2 iff the receiving processor n_2 has enough free capacity, that is, if $\sigma_d \leq f(n_2, s)$. The transmission then incurs a cost of $T(n_1, d, n_2)$. That is, if $s[\![t(n_1, d, n_2)]\!]$ is the state resulting from this transmission, then $s[\![t(n_1, d, n_2)]\!](n_2) = s(n_2) \cup \{d\}$, *ceteris paribus*. Finally, it is always possible for processor n to delete some data chunk d. The respective action $del(n, d)$ incurs no cost, and $s[\![del(n, d)]\!](n) = s(n) \setminus \{d\}$, *ceteris paribus*.

Given a DPPS task $\langle D, N, A, T, s_0, G \rangle$, its set of operators is $O := A \cup \{t(n_1, d, n_2) \mid T(n_1, d, n_2) < \infty\} \cup \{del(n, d) \mid n \in N, d \in D\}$, and $|O|$ is polynomial in $|D|$, $|A|$, and $|N|$. A sequence of operator instances $\pi = \langle o_1 \dots o_m \rangle$ is applicable if o_i is applicable in s_{i-1}, where s_0 is the initial state, and $s_i := s_{i-1}[\![o_i]\!]$. The cost of π is the sum of its operator costs, and it is a solution to the task if s_m satisfies the goal, that is, if $G(n) \subseteq s_m(n)$ for all $n \in N$. Finally, π is optimal if its cost is minimal among all the solutions to the task.

Expressivity and Complexity

We now turn to consider the expressivity and computational complexity of DPPS. Despite the fact that the syntax of DPPS seems very different from SQL-style database query languages, we begin by showing that DPPS is strictly more expressive than relational algebra (Codd 1970) extended with aggregate functions (Klug 1982). Although there is a shift in data-parallel computing systems towards the NoSQL paradigm (Padhy, Patra, and Satapathy 2011), relational algebra is still a popular formalism for describing queries. Furthermore, modern data-parallel processing systems can impose an "ad-hoc" schema on non-relational data, thus making relational algebra still relevant to these systems. It is also worth noting at this point that DPPS is not limited to relational algebra, and can be used with other data models, that may be more suited to the NoSQL paradigm.

Presenting a complete description of relational algebra here is infeasible, but we will briefly mention that a relational algebra query can be represented as a tree, whose leaves are either constants or relations (i.e., database tables), and whose inner nodes are projection (π), selection (σ), or aggregation ($\mathcal{A}$) of a single child node, or cross product ($\times$), set union ($\cup$) or set difference ($\setminus$) of two child nodes. Similarly to the alias mechanism of SQL, we assume wlog that each occurrence of a relation name in that tree is assigned a distinct name.

Theorem 1 *Given an extended relational algebra query Φ, we can efficiently construct a solvable* DPPS *task Π such that any solution π for Π induces a valid query plan φ for Φ.*

The detailed proofs of the formal claims are relegated to a technical report. However, we attempt to provide the main ideas behind the proofs, especially if these carry some helpful insights. For the proof of Theorem 1, we construct a DPPS task Π with a single processor n. The possible data chunks D of Π are all subexpressions of Φ, and they are constructed from the tree representation of Φ by taking, for each node e in Φ, the subtree rooted at e, that is $D := \{\text{subtree}(e) \mid e \in \Phi\}$. As the data chunks are represented in Φ explicitly, their number is linear in the size of Φ.

The computation primitives A of Π correspond to the internal nodes of Φ; for each internal node e we construct a primitive a_e, whose inputs are the children of e and whose output is e, with a cost that reflects the (typically estimated) execution cost of the computation. In the initial state of Π, the single processor n contains the data chunks corresponding to all leaf nodes, and the goal is for n to hold the data chunk corresponding to Φ. Since there is only one processor in Π, the transmission cost function T is irrelevant. It is easy to see that all solutions to Π contain the same computations, one for each internal node in Φ (the only choice the solver makes is the order in which these computations are performed), and all these solutions of Π correspond to query plans for Φ.

This proof of Theorem 1 is simple, but its construction restricts the scope of feasible query plans. However, there is a simple fix for that, which we describe here in informal terms. First, we add computation primitives corresponding to relational algebra equivalence rules, with a cost of 0. For example, the equivalence rule $\sigma_p(\sigma_q(X)) = \sigma_q(\sigma_p(X))$ induces the computation which takes as input a data chunk of the form $\sigma_p(\sigma_q(X))$, and produces as output the data chunk $\sigma_q(\sigma_p(X))$. Of course, this means we must extend the set of possible data chunks D to include all possible expressions derived in this manner. However, as noted earlier, we do not have to explicitly enumerate all possible data chunks, but can define them implicitly, by the base relations and the relational algebra operators. Additionally, we can add operators corresponding to "macros", when these computations can be more efficiently executed. For example, joins can be expressed as selection over a cross-product of two relations, but are usually much more efficient to execute than first performing a cross-product, and then a selection over its output. These allow the solver to find a more efficient plan, while using the equivalence computations to prove that the result is equivalent to the original query.

In essence, this finalizes an encoding of (non-distributed) query optimization as a DPPS task. However, our main motivation comes from the need to perform *distributed* computations. While the basic ideas described above still work, there are several more issues that need to be dealt with. First, a typical distributed database will fragment tables, and thus processors will no longer contain only base relations (and constants) in the initial state, but will usually contain some table fragments, which can be expressed as selection and/or projection of the base relations. For example, assuming we have two processors which store some table T by hash-partitioning on field f, in the initial state processor n_0 will hold $\sigma_{\text{hash}(f)=0}(T)$ and processor n_1 will hold $\sigma_{\text{hash}(f)=1}(T)$. We must also make sure to include equivalence rules which allow us to merge such data chunks, i.e., the equivalence rule $\sigma_{true}(X) = X$, along with $\sigma_\phi(X) \cup \sigma_\psi(X) = \sigma_{\phi \vee \psi}(X)$, and the fact that $(\text{hash}(f) = 1) \vee (\text{hash}(f) = 0) \equiv true$, which is easily extended for any partition.

Finally, we remark that a computing cluster, as well as the data stored and accessible on that cluster, are resources that are usually shared between multiple users, and thus must be used to satisfy all users' needs. It has already been noted that the overall system performance could benefit from sharing computations between different queries. For example, the Comet system (He et al. 2010) constructs an execution plan for a series of DryadLINQ queries, and cross program optimization between Pig Latin programs is described by Olston et al. (2008a). With the DPPS formalism, multi-goal optimization is trivial, as the formalism already supports the specification of multiple goals.

Given the expressivity of DPPS, it is not surprising that optimal data-parallel program synthesis is computationally hard. More importantly, it turns out that this problem remains NP-hard even under severe restrictions, because the computational hardness of DPPS stems from numerous sources. In fact, it turns out that even the satisficing variant of this problem is NP-hard.

Theorem 2 *Satisficing data-parallel program synthesis is NP-hard, even when the possible data chunks are given explicitly.*

The proof of Theorem 2 is by reduction from 3SAT; the induced DPPS tasks have processors with only a fixed memory capacity, and the solution must carefully manage the memory resource. The following two theorems show that *optimal* data-parallel program synthesis is NP-hard even under severe restrictions.

Theorem 3 *Optimal data-parallel program synthesis with a single processor is NP-hard, even if the possible data chunks are given explicitly, and there are no memory constraints.*

Theorem 4 *Optimal data-parallel program synthesis with a single data chunk is NP-hard.*

The proof of Theorem 3 is by polynomial reduction from optimal delete-free STRIPS planning (Bylander 1994), while the proof of Theorem 4 is by polynomial reduction from the minimum Steiner tree in a graph problem (Karp 1972; Garey and Johnson 1979). These proofs capture two sources of complexity of DPPS: The complexity in Theorem 3 stems from the number of possible data chunks, while Theorem 4 captures the complexity which stems from the network structure.

Although both optimal and satisficing DPPS are worst-case intractable in many cases, that does not mean that none of their practically relevant fragments is polynomial time. While much further investigation is needed here, Theorem 5 already captures one such fragment of tractability, which in particular contains our multiple histogram running example.

Theorem 5 *Satisficing data-parallel program synthesis with no memory constraints can be solved in polynomial time, when the possible data chunks are given explicitly.*

The proof is by polynomial reduction to satisficing delete-free STRIPS planning, which is known to be polynomial-time solvable (Bylander 1994).

DPPS meets STRIPS

The worst case hardness of data-parallel program synthesis is clearly not something to be ignored, but obviously it does not imply that solving practical problems of interest is out of reach. The two options here would be either to develop a special-purpose solver for DPPS, or to compile DPPS tasks into a canonical combinatorial search problem, and use an off-the-shelf solver for the latter. The key advantage of the second option, which we consider here, is that using off-the-shelf solvers only requires modeling the problem of interest in the right formalism, without programming. Furthermore, these solvers tend to be better engineered and more robust than special-purpose solvers for niche problems. Since DPPS is all about synthesizing (possibly partially ordered) goal-achieving sequences of actions, a natural such target of compilation for DPPS would be this or another standard formalism for deterministic planning, with STRIPS probably being the most canonical such formalism (Fikes and Nilsson 1971).

A STRIPS planning task with action costs is a 5-tuple $\Pi = \langle P, O, \mathbb{C}, s_0, G \rangle$, where P is a set of propositions, O is a set of actions, each of which is a triple $o = \langle \mathsf{pre}(o), \mathsf{add}(o), \mathsf{del}(o) \rangle$, $\mathbb{C} : O \to \mathbb{R}^{0+}$ is the action cost function, $s_0 \subseteq P$ is the initial state, and $G \subseteq P$ is the goal. An action o is applicable in state s if $\mathsf{pre}(o) \subseteq s$, and if applied in s, results in the state $s' = (s \setminus \mathsf{del}(o)) \cup \mathsf{add}(o)$. A sequence of actions $\langle o_0, o_1, \ldots, o_n \rangle$ is applicable in state s_0 if o_0 is applicable in s_0 and results in state s_1, o_1 is applicable in s_1 and results in s_2, and so on. The cost of an action sequence is the sum of the action costs, $\sum_{i=0}^{n} \mathbb{C}(o_i)$. The state resulting from applying action sequence π in state s is denoted by $s[\![\pi]\!]$. An action sequence $\langle o_0, o_1, \ldots, o_n \rangle$ is a plan for Π if $G \subseteq s_0[\![\langle o_0, o_1, \ldots, o_n \rangle]\!]$, and it is an optimal plan if no cheaper plan exists.

We begin by describing a straightforward compilation of DPPS tasks with *explicitly* specified possible data chunks and no memory constraints. Given a DPPS task with processors N, data chunks D, computations A, and without memory constraints, we show how to construct a STRIPS task $\Pi = \langle P, O, \mathbb{C}, s_0, G \rangle$, such that there is a cost-preserving one-to-one correspondence between solutions for Π and solutions to the DPPS task. The propositions are $P = \{holds(n, d) \mid n \in N, d \in D\}$, and they describe which processor currently holds which data chunk. The operators of Π are given by $O = \{transmit(n_1, d, n_2), delete(n_1, d), compute(n_1, a) \mid n_1, n_2 \in N, d \in D, a \in A\}$. The transmit operators are described by $transmit(n_1, d, n_2) = \langle\{holds(n_1, d)\}, \{holds(n_2, d)\}, \emptyset\rangle$, with a cost of $T(n_1, d, n_2)$. The delete operators are described by $delete(n_1, d) = \langle\{holds(n_1, d)\}, \emptyset, \{holds(n_1, d)\}\rangle$, with a cost of 0. Finally, for $a = \langle\bar{I}, \bar{O}, C\rangle$, the compute operators are described by $compute(n_1, a) = \langle\{holds(n_1, d) \mid d \in \bar{I}\}, \{holds(n_1, d) \mid d \in \bar{O}\}, \emptyset\rangle$, with a cost of $C(n_1)$.

It is not hard to verify that any solution for the planning task Π is also a solution to the given DPPS task, and that the cost of the solutions is the same. Note that the only reason we require that the DPPS task does not have any memory constraints is that STRIPS does not support numerical variables, and so we cannot formulate the requirement that a processor has enough free memory to store any data chunks it computes or receives by transmission. However, numerical extensions to STRIPS do exist (Fox and Long 2003), and we can use such a numerical planning framework to also pose memory capacity constraints.

The more substantial issue remains the restriction to explicitly specified possible data chunks, because typically we should expect the data chunks to be specified only implicitly. This issue has been addressed already in the planning literature, and in particular, in the context of the DPADL action language (Golden 2002). Here, however, we are interested in using off-the-shelf STRIPS planners, and fortunately, this issue can be overcome in STRIPS as well. Below we demonstrate this by describing the DPPS task that encodes relational algebra query optimization, which was described informally in the previous section. The propositions we use here encode a relational algebra expression tree. The objects, i.e., the parameters to predicates and operators, correspond to nodes in a relational algebra expression tree, as well as selection predicates, field lists, and aggregation functions. We do not give a full description here for the sake of brevity, but illustrate the main points.

First, we have propositions that describe the structure of the relational algebra expression tree. Each node in the tree has an expression type, and each type of node has either one subexpression (select, project, and aggregation) or two subexpression (cross-product, union, and difference). Additionally, select nodes also have a selection predicate, project nodes have a list of fields, and aggregation nodes have a list of fields and an aggregation function. For example, the proposition $select(e, p, e_1)$, indicates that node e is the result of applying selection predicate p on node e_1, or in relational algebra notation $e = \sigma_p(e_1)$. Another example is $crossproduct(e, e_1, e_2)$ which indicates that e is the result of the cross-product of e_1 and e_2, or $e = e_1 \times e_2$. The full list of predicates, with their interpretation in relational algebra, is given in Table 1.

A second type of proposition we use is $clear(e)$, which indicates that e has not already been set, that is, that e is currently not part of any relational algebra expression. Additionally, the proposition $eqv(e_1, e_2)$ indicates that the expressions represented by e_1 and e_2 are equivalent. Finally, as in the previous construction, we have the proposition

Proposition	RA Interpretation
$select(e, p, e_1)$	$e = \sigma_p(e_1)$
$project(e, f, e_1)$	$e = \pi_f(e_1)$
$aggregate(e, f, a, e_1)$	$e = \mathcal{A}_{f,a}(e_1)$
$crossproduct(e, e_1, e_2)$	$e = e_1 \times e_2$
$union(e, e_1, e_2)$	$e = e_1 \cup e_2$
$diff(e, e_1, e_2)$	$e = e_1 \setminus e_2$

Table 1: STRIPS propositions and their interpretation in relational algebra

$holds(n, d)$ for every $n \in N$ and every relational algebra expression node d, which indicates that processor n hold the data chunk d.

We also construct three types of operators. First, we have the transmission and deletion operators, which are part of any DPPS task, as described previously. As before, the cost of the transmission operators stems from the DPPS task's transmission cost function, and the cost of the delete operators is 0.

Second, we have operators corresponding to actual relational algebra operations: selection, projection, aggregation, cross-product, union, and difference. The cost of these operators reflects the estimated execution cost. Each such operator is also parametrized by which processor performs the computation. For example, the operator $doselect(n, e_1, p, e_2)$ requires $clear(e_1)$ and $holds(n, e_2)$, adds the propositions $select(e_1, p, e_2)$ and $holds(n, e_1)$, and deletes $clear(e_1)$. As before, operators representing relational algebra "macros", such as join, can also be encoded. For example, an operator joining relations e_1 and e_2 over predicate p creates two nodes: one representing $e_1 \times e_2$, and another representing $\sigma_p(e_1 \times e_2)$. However, the cost of this operator is the cost of performing the join, rather than the cost of doing the full cross-product and selection.

Finally, we have operators corresponding to equivalence rules. The cost of these operators is zero, because, as noted earlier, they are not actual computations, but are rather used by the solver to "prove" that its solution is correct. These operators are all encoded so that they construct a new relational algebra expression node — the "input" is never modified — and add the proposition that indicates that these expression are equivalent. For example, the commutativity of selection, that is equivalence rule $\sigma_p(\sigma_q(X)) = \sigma_q(\sigma_p(X))$ is encoded by the operator described here:

$commute\text{-}select(SpSqX, SqX, X, SqSpX, SpX)$	
pre	$select(SqX, q, X) \wedge select(SpSqX, p, SqX) \wedge clear(SqSpX) \wedge clear(SpX)$
add	$select(SpX, p, X) \wedge select(SqSpX, q, SpX) \wedge eqv(SpSqX, SqSpX)$
del	$clear(SpX) \wedge clear(SqSpX)$

We can always encode these equivalence rules as operators with a fixed number of parameters, because the equivalence rules are local in nature, in the sense that they do not look deeper than one or two levels into the input relational algebra expression tree. One important point to note is that STRIPS does not support the creation of new objects. There-

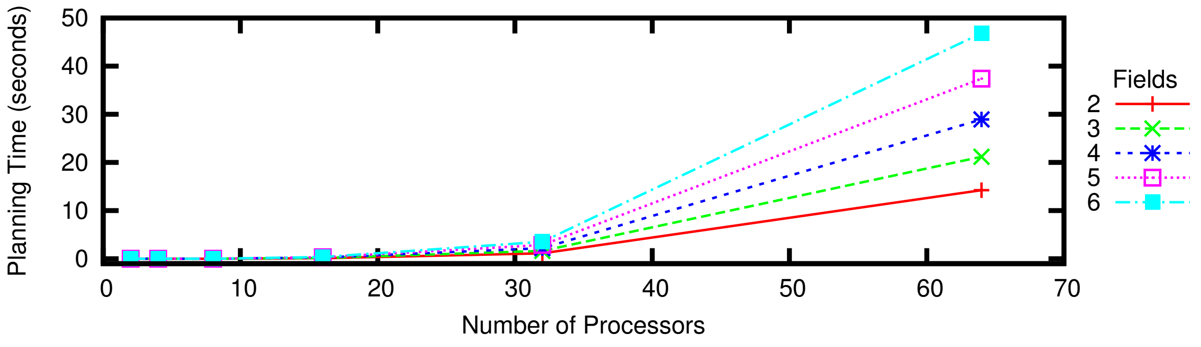

Figure 1: Runtime for obtaining f histograms over different fields from the same table, fragmented across n processors

fore, the number of relational algebra expression nodes that are used in the solution must be set before planning begins. However, an algorithm which iteratively increases the number of nodes can be used to overcome this limitation.

Theorem 6 *Let Π be a* DPPS *task, and $k \in \mathbb{N}$. There exists a* STRIPS *task Π' that can be constructed from Π in time polynomial in $|\Pi|$ and k, such that, if Π has a solution consisting of at most k operators, then*

(i) *Π' is solvable, and*

(ii) *every plan π' for Π' induces a solution π for Π of the same cost.*

In order to provide some empirical evidence that our approach is practical, we have encoded a DPPS task which describes our running example of multiple histogram computations over a single table, as a planning task. We assume the table is partitioned by its primary key across a cluster with n processors, and that the user wants a histogram of the table by f different fields. The computational operators here are $count(d_i)$ which generates f partial histograms, one by each field, from table fragment d_i, and $merge(h_i)$ which requires all n partial histograms according to field i, and merges them into a histogram of the full table by field i. We did not model memory constraints, and so this domain is delete-free.

We varied n from 2 to 64 and f from 2 to 6, and solved these planning tasks with the Fast Downward planner (Helmert 2006) using greedy best first search with the FF heuristic (Hoffmann and Nebel 2001). Figure 1 shows the total planning time for these different values, with a separate line for each value of f. The hardest planning problem, of computing a histogram by 6 different fields across a cluster with 64 processors, was still solved in under a minute. Note that we did not implement the operators *execution*, and so we can not compare to actual distributed DBMS solutions, for which we can only measure query execution time. Finally, we remark that although the planner we used does not guarantee optimal solutions, in this case all the solutions that were obtained were optimal, and scanned the table only once.

Summary

We have described a formal model of data-parallel program synthesis, DPPS, which generalizes the specific data processing systems developed in the area of data-parallel computing. The key advantage of working at the level of the DPPS model is that it is easy to separate the modeling of the domain (i.e., the data chunks and computations) from the specific task (i.e., the network topology, current state of the system, and current queries of the users). The domain modeling could be done by the software development team, by symbolically annotating user-defined operators. The specific task is generated when a query is given to the system by a user, who need not be aware of implementation techniques or network topology. This allows the software development team to focus on optimizing individual low-level functions, while the system automatically generates an optimized query plan.

DPPS is more expressive than relational algebra with aggregate functions, allowing for both taking into account the distributed nature of data-parallel computing, as well as incorporating arbitrary user-defined operators of the form supported by the current data-parallel computing systems. The expressivity of DPPS makes reasoning in it computationally hard in the worst case, and we discuss various sources of this time complexity. Beyond the worst-case complexity analysis, we showed how DPPS tasks can be compiled into STRIPS, a standard formalism for deterministic action planning. Using a canonical histogram computation example, we demonstrated how one can use off-the-shelf STRIPS planning tools to solve such DPPS tasks.

In terms of future work, examining the relationship between tradional measures for query complexity from the DB community, and the complexity of the corresponding planning task could lead to cross-fertilization between the fields. Additionally, studying the relationship between tractable fragments of DPPS and tractable fragments of planning could also lead to some interesting results.

Acknowledgements

This work was carried out in and supported by the Technion-Microsoft Electronic-Commerce Research Center. We thank the anonymous reviewers for their helpful comments.

References

Ambite, J. L., and Knoblock, C. A. 2001. Planning by rewriting. *JAIR* 15:207–261.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *AIJ* 69(1–2):165–204.

Cafarella, M. J., and Ré, C. 2010. Manimal: Relational optimization for data-intensive programs. In *Proceedings of the 13th International Workshop on the Web and Databases 2010*.

Chaiken, R.; Jenkins, B.; Larson, P.-A.; Ramsey, B.; Shakib, D.; Weaver, S.; and Zhou, J. 2008. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.* 1(2):1265–1276.

Chien, S. A.; Fisher, F.; Lo, E.; Mortensen, H.; and Greeley, R. 1999. Using artificial intelligence planning to automate science image data analysis. *Intelligent Data Analysis* 3(3):159–176.

Codd, E. F. 1970. A relational model of data for large shared data banks. *CACM* 13(6):377–387.

Cohen, S. 2006. User-defined aggregate functions: bridging theory and practice. In *Proc. SIGMOD 2006*, 49–60. ACM.

Dean, J., and Ghemawat, S. 2008. MapReduce: simplified data processing on large clusters. *CACM* 51(1):107–113.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *AIJ* 2:189–208.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *JAIR* 20:61–124.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability — A Guide to the Theory of NP-Completeness*. Freeman.

Golden, K. 2002. DPADL: An action language for data processing domains. In *Proceedings of the 3rd NASA Intl. Planning and Scheduling workshop*, 28–33.

He, B.; Yang, M.; Guo, Z.; Chen, R.; Su, B.; Lin, W.; and Zhou, L. 2010. Comet: batched stream processing for data intensive distributed computing. In *Proc SoCC 2010*, 63–74.

Helmert, M. 2006. The Fast Downward planning system. *JAIR* 26:191–246.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

Isard, M., and Yu, Y. 2009. Distributed data-parallel computing using a high-level programming language. In *Proc. SIGMOD 2009*, 987–994. ACM.

Isard, M.; Budiu, M.; Yu, Y.; Birrell, A.; and Fetterly, D. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. SIGOPS/Eurosys 2007*, 59–72. ACM.

Karp, R. M. 1972. Reducibility among combinatorial problems. In Miller, R. E., and Thatcher, J. W., eds., *Complexity of Computer Computations*. Plenum Press. 85–103.

Klug, A. 1982. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *JACM* 29(3):699–717.

Olston, C.; Reed, B.; Silberstein, A.; and Srivastava, U. 2008a. Automatic optimization of parallel dataflow programs. In *USENIX 2008*, 267–273.

Olston, C.; Reed, B.; Srivastava, U.; Kumar, R.; and Tomkins, A. 2008b. Pig latin: a not-so-foreign language for data processing. In *Proc. SIGMOD 2008*, 1099–1110. ACM.

Ozsu, M. T., and Valduriez, P. 2007. *Principles of Distributed Database Systems*. Upper Saddle River, NJ, USA: Prentice Hall Press, 3rd edition.

Padhy, R. P.; Patra, M. R.; and Satapathy, S. C. 2011. RDBMS to NoSQL: Reviewing Some Next-Generation Non-Relational Database's. *International Journal of Advanced Engineering Science and Technologies* 11(1).

Thusoo, A.; Sarma, J. S.; Jain, N.; Shao, Z.; Chakka, P.; Anthony, S.; Liu, H.; Wyckoff, P.; and Murthy, R. 2009. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.* 2(2):1626–1629.

Thusoo, A.; Sarma, J. S.; Jain, N.; Shao, Z.; Chakka, P.; Zhang, N.; Antony, S.; Liu, H.; and Murthy, R. 2010. Hive - a petabyte scale data warehouse using hadoop. In *Proc. ICDE 2010*, 996 – 1005.

Ullman, J. D. 1988. *Principles of Database and Knowledge-Base Systems. Volume I: Classical Database Systems*. Computer Science Press.

Valduriez, P. 1993. Parallel database systems: Open problems and new issues. *Distributed and Parallel Databases* 1(2):137–165.