The VLDB Journal (2017) 26:177–201
DOI 10.1007/s00778-016-0446-1

REGULAR PAPER

In-database batch and query-time inference over probabilistic graphical models using UDA–GIST

Kun Li[1] · **Xiaofeng Zhou**[1] · **Daisy Zhe Wang**[1] · **Christan Grant**[2] · **Alin Dobra**[1] · **Christopher Dudley**[1]

Received: 15 December 2015 / Revised: 22 September 2016 / Accepted: 20 October 2016 / Published online: 2 November 2016

Abstract To meet customers' pressing demands, enterprise database vendors have been pushing advanced analytical techniques into databases. Most major DBMSes use user-defined aggregates (UDAs), a data-driven operator, to implement analytical techniques in parallel. However, UDAs alone are not sufficient to implement statistical algorithms where most of the work is performed by iterative transitions over a large state that cannot be naively partitioned due to data dependency. Typically, this type of statistical algorithm requires pre-processing to set up the large state in the first place and demands post-processing after the statistical inference. This paper presents general iterative state transition (GIST), a new database operator for parallel iterative state transitions over large states. GIST receives a state constructed by a UDA and then performs rounds of transitions on the state until it converges. A final UDA performs post-processing and result extraction. We argue that the combination of UDA and GIST (UDA–GIST) unifies data-parallel and state-parallel processing in a single system, thus significantly extending the analytical capabilities of DBMSes. We exemplify the framework through two high-profile batch applications: cross-document coreference, image denoising and one query-time inference application: marginal inference queries over probabilistic knowledge graphs. The 3 applications use probabilistic graphical models, which encode complex relationships of different variables and are powerful for a wide range of problems. We show that the in-database framework allows us to tackle a 27 times larger problem than a scalable distributed solution for the first application and achieves 43 times speedup over the state-of-the-art for the second application. For the third application, we implement query-time inference using the UDA–GIST framework and apply over a probabilistic knowledge graph, achieving 10 times speedup over sequential inference. To the best of our knowledge, this is the first in-database query-time inference engine over large probabilistic knowledge base. We show that the UDA–GIST framework for data- and graph-parallel computations can support both batch and query-time inference efficiently in databases.

Keywords In-database analytics · Query-time inference · Batch inference · Data-parallel analytics · Graph-parallel analytics

Kun Li and Xiaofeng Zhou both authors contribute equally to this paper.

✉ Xiaofeng Zhou
xiaofeng@cise.ufl.edu

Kun Li
kli@cise.ufl.edu

Daisy Zhe Wang
daisyw@cise.ufl.edu

Christan Grant
cgrant@ou.edu

Alin Dobra
adobra@cise.ufl.edu

Christopher Dudley
cdudley@cise.ufl.edu

[1] Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611, USA

[2] School of Computer Science, University of Oklahoma, Norman, OK 73019, USA

1 Introduction

With the recent boom in big data analytics, many applications require large-scale data processing as well as advanced statistical methods such as random walk and MCMC algorithms.

Connecting tools for data processing (e.g., DBMSes) and tools for large-scale machine learning (i.e., GraphLab [27,28]) using a system-to-system integration has severe limitations including inefficient data movement between systems, impedance mismatch in data representation and data privacy issues [25,47]. In the database community, there is a renewed interest in integrating statistical machine learning (SML) algorithms into DBMSes [18]. Such integration allows both SQL-based data processing and statistical data analytics, providing a full spectrum of solutions for data analytics in an integrated system.

Most SML algorithms can be classified into two classes in terms of parallel execution. The first well-studied class of SML algorithms require multiple iterations of the same data. Such SML methods include Linear Regression, K-means and EM algorithms, which can be parallelized within each iteration using naive data partitioning. The overall algorithms can be driven by an outside iteration loop. The parallel implementation of this class of SML algorithms is supported in MADlib [11,18] and Mahout [29]. Most commercial databases incorporate the support for such *data-parallel* SML algorithms in the form of UDAs with iterations in external scripting languages.

A second class of SML algorithms involve pre-processing and constructing a large state with all the data. The state space cannot be naively partitioned, because the random variables in the state are correlated with each other. After the state is built, the algorithms involve iterative transitions (e.g., sampling, random walk) over the state space until a global optimization function converges. Such operations are computation intensive without any data flow. After convergence is reached, the state needs to be post-processed and converted into tabular data. We dub this class of SML algorithms *state-parallel* algorithms, where the states can be graphs, matrices, arrays or other customized data structures. Examples of this type of SML algorithms include MCMC and belief propagation algorithms.

Several significant attempts have been made toward efficient computation frameworks for SML both in MPP databases such as MADlib [11,18] and in other parallel and distributed frameworks such as Mahout [29], GraphLab [27,28] and GraphX [54]. However, no previous work can efficiently support both data-parallel and state-parallel processing in a single system, which is essential for many new applications that applies SML algorithms over large amounts of data. To support such advanced data analytics, the UDA–GIST framework developed in this work unifies data-parallel and state-parallel processing by extending existing database frameworks.

Graph-parallel algorithms are a special type of state-parallel algorithm whose state is an immutable graph. Examples of graph-parallel algorithms include inference over large probabilistic graphical models [23] encoding complex relationships between variables and expressively powerful, such as Bayesian Networks [17] and Markov Random Fields [38], where the graph-based state can have hundreds of millions of nodes and billions of edges. While parallel DBMSes and Map-Reduce frameworks cannot efficiently express graph-parallel algorithms, other solutions exist such as GraphLab [27,28] and GraphX [54], both of which have graph-based abstractions. These graph-parallel systems simplify the design and implementation of algorithms over sparse graphs using a high-level abstraction, but they miss the opportunity of using more efficient data structures to represent the state space of a complete/dense graph, a matrix or a dynamic graph. For example, if the state is a matrix, representing it as a generalized graph can make the state building orders of magnitude slower and hamper inference significantly due to worse access pattern over a generalized graph. Moreover, GraphLab does not support data-parallel processing for state construction, post-processing, tuple extraction and querying. As shown in the experiments of this paper, it is time-consuming to build the state, to post-process and to extract the results. The combined time for these steps can exceed the inference time.

In this paper we ask and positively answer a fundamental question: *Can SML algorithms with large state transition be efficiently integrated into a DBMS to support data analytics applications that require both data-parallel and state-parallel processing?* Such a system would be capable of efficient state construction, statistical inference, post-processing and result extraction.

The first challenge to support efficient and parallel large iterative state transition in-database is the fact that DBMSes are fundamentally data-driven, i.e., computation is tied to the processing of *tuples*. However, iterative state transition-based algorithms are computation driven and dissociated from tuples. Supporting such computation needs additional operator abstraction, task scheduling and parallel execution in a DBMS.

To solve the first challenge of DBMS computation being tied to tuples, we introduce an abstraction that generalizes GraphLab API called *Generalized Iterative State Transition* (GIST). GIST requires the specification of an inference algorithm in the form of four abstract data types: (1) the GIST State representing the state space; (2) the Task encoding the state transition task for each iteration; (3) the Scheduler responsible for the generation and scheduling of tasks; and (4) the convergence UDA (cUDA) ensuring that the stopping condition of the GIST operation gets observed.

The second challenge is that the state has to be represented efficiently inside the DBMS, compatible with the relational data model. Large memory may be required for large states during state transition, and new state transition operations have to be efficiently integrated into an existing DBMS.

To solve the second challenge of efficient representation inside DBMS, we implement and integrate the GIST operator into a DBMS along with user-defined functions (UDFs) [3] and user-defined aggregates (UDAs) [48]. The efficient GIST implementation is achieved using the following techniques: (1) asynchronous parallelization of state transition; (2) efficient and flexible state implementation; (3) lock-free scheduler. The key of an efficient integration between the non-relational GIST operator and a relational DBMS engine is to use UDAs to build large states from DBMS tuples and to post-process and extract the result tuples from GIST.

The UDA–GIST framework can support a large class of advanced SML-based applications where both data-driven computation and large state transition are required. The specific contributions we make with UDA–GIST framework are:

- *We propose a general iterative state transition (GIST) operator abstraction for implementing state-parallel SML algorithms.* We provide insights and details into how a high performance implementation of GIST can be obtained in a DBMS.
- *We explain how a GIST operator implementing the abstraction can be efficiently integrated as a first-class operator in a DBMS.* The deep integration of GIST and UDA results in the UDA–GIST framework. We intend the framework to be general for most SML algorithms with support for both data-parallel and state-parallel computations. Compared with GraphLab, the framework trades off implementation complexity for expressiveness and performance. While the application developers may need to implement their own scheduler for synchronization and deadlock resolution, they are given the flexibility to specify their own state representation and parallel execution strategy, which as shown in our experiments can achieve orders-of-magnitude performance gain. The main merits of the UDA–GIST framework are: (1) building state, post-processing and extracting results in parallel; (2) unifying data-parallel and state-parallel computations in a single system; (3) representing states using more compact application-specific data structures; (4) implementing application-specific scheduler for higher degree of parallel execution; (5) providing an efficient mechanism to detect global/local convergence.
- *We exemplify the use of the UDA–GIST abstraction by implementing three representative SML algorithms and applications*: *Metropolis–Hastings algorithm* [2] for cross-document coreference, *loopy belief propagation* [33] for image denoising and *Gibbs Sampling* [6] for query-time marginal inference. We show that the applications can be executed using the extended DBMS execution engine with the UDA–GIST framework. These three applications exemplify the efficiency of the UDA–GIST framework for large classes of state-parallel SML methods such as Markov-chain Monte Carlo and message passing algorithms.
- *We show that UDA–GIST framework results in orders-of-magnitude speedup for the three exemplifying applications comparing with state-of-the-art systems.* For the first application, using a similar coreference model, features and dataset as described in a recent effort at Google [44], the UDA–GIST system achieves comparable results in terms of accuracy in 10 min over a multi-core environment, while the Google system uses a cluster with 100–500 nodes. Results show that this UDA–GIST system can also handle a 27 times larger dataset for coreference. For the second application, we show that UDA–GIST outperforms GraphLab's implementation of image denoising with loopy belief propagation by three orders of magnitude for state building and post-processing, and up to 43 times in overall performance. For the third application, we can achieve one order of magnitude of speedup over the sequential implementation.

This paper is an extension to the UDA–GIST [26] paper, providing in-database query-time inference over probabilistic graphical model to unify in-database batch and query-time inference using UDA–GIST framework.

To exemplify query-time inference, we implement probabilistic marginal inference over probabilistic knowledge graph. However, inference over large probabilistic knowledge graph is prohibitive, and existing systems [9,35] are primarily designed for batch processing, not suited for interactive queries. Motivated by the scalability issue of sampling methods and lack of efficient query-time inference over probabilistic knowledge graph, we implement the sampling methods in UDA–GIST framework to speed up the inference. However, real-time response still cannot be achieved due to the sheer size of the graph. Thus we apply a k-hop approach to extract the k-hop network, centered around the query node, to approximate the marginal probability. The intuition behind the k-hop approach is that the farther away the node is from the query node, the less influence the node has on the query node. By selecting the k-hop network, the inference is focused on the important selection to query node. User can specify the number of hops to achieve the trade-off between accuracy and time. The approximation error decreases as the number of hops increases. Results show that it can achieve accurate results with negligible error at query time (Figs. 14, 15).

Specifically, the additional contributions we make in this paper are:

- We present the k-hop query inference to approximate the inference over whole factor graph, which provides a trade-off between time and accuracy. The experiment

results show the k-hop approach achieves orders of magnitude of speedup while achieving good approximation with small number of hops in real datasets.
– We parallelize the Gibbs sampling using UDA–GIST framework. The results show an order of magnitude speedup over single-threaded Gibbs sampling.

In the rest of this paper, we first give a system overview in Sect. 2, and present GIST API in Sect. 3. Then we introduce the three applications, including the background knowledge and algorithms, and UDA–GIST implementation of the algorithms in Sects. 4, 5 and 6, respectively. Finally, we show that the GIST and an efficient integration with DBMS systems result in orders-of-magnitude performance gain in Sect. 7.

2 System overview

As we explained in the introduction, state-parallel SML algorithms involve iterative state transitions over a large state space. The execution pipeline for such SML tasks is shown in Fig. 1. UDAs are used to construct the in-memory state from the database tables. The GIST takes the state from the UDA and performs iterative state transition over the shared state. The cUDA inside the GIST is used to evaluate the convergence of the state. Finally, the UDA Terminate function to the right of GIST operator supports post-processing and converting the converged final state into relational data.

We design an extended DBMS system architecture that supports the GIST operator and GIST execution engine for iterative state transition over large state space based on a shared-memory paradigm. GIST operators together with data-driven operators in DBMSes such as SQL queries and UDAs can provide efficient and scalable support for a wide spectrum of advanced data analysis pipelines based on SML models and algorithms.

As shown in Fig. 2, the GIST operators are implemented as first-class citizens, similar as UDAs in an DBMS. UDAs and GIST operators are implemented using two different APIs and are supported by two different execution models. In this paper, the data processing is performed over multi-core machines. Different inference algorithms can be implemented using the GIST and UDA APIs, including loopy belief propagation (LBP) and Markov-chain Monte–Carlo (MCMC) algorithms. Using such inference algorithms, different statistical models can be supported to develop applications such as image denoising and cross-document coreference.

The UDA–GIST framework expands the space of feasible problems on one single multi-core machine and raises the bar on required performance for a complicated distributed system. As an example, our experiments for the coreference application use a 27 times larger dataset than the state-of-the-art in a distributed cluster with 100–500 nodes [44]. One premise of this work is that a single multi-core server is equipped with hundreds of gigabytes of memory, which is sufficiently big to hold the states of most applications. Second, a multi-core server is inexpensive to install, administer and is power efficient. It is hard to acquire and maintain a cluster with hundreds of nodes.

The UDAs follow the traditional API, consisting of three functions: `Init()`, `Accumulate()` and `Merge()`. The

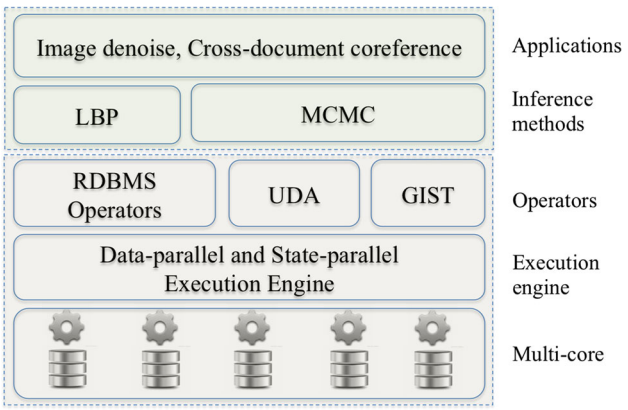


Fig. 2 An extended DBMS architecture to support data-parallel and state-parallel analytics

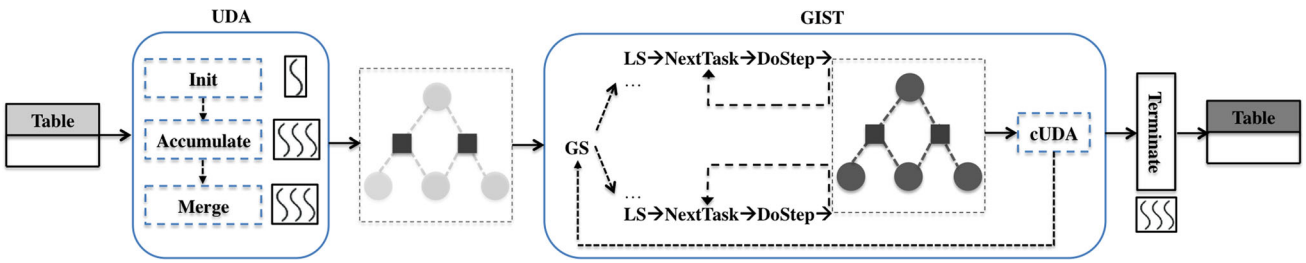


Fig. 1 ML Pipeline with large state transition. UDAs are used to construct the in-memory state from the database tables. The GIST takes the state from the UDA and performs computation over the shared state. The cUDA inside the GIST is used to evaluate the convergence of the state. The UDA Terminate function to the right of GIST supports converting the state into relational data

`Init` is used to set up the state appropriately before any computation begins. It is similar to a constructor in many high-level languages. `Accumulate` takes a tuple as input and adds the tuple to the state that it is maintaining. Tuples can be read from a database or files in the local file system. `Merge` combines the state maintained by two UDA instances of the same type. It must be associative, so that states do not need to be merged in any particular order for computational efficiency.

3 General iterative state transition (GIST)

Compared to the framework proposed by GraphLab [27], GIST API supports more general data structure to represent the state and supports more flexible scheduler for parallel execution. While, by design, GraphLab supports only immutable graph-based data structures, we design GIST to support general data structures to represent large state spaces, including arrays, matrices, and static/dynamic graphs. In addition, we further generalize GraphLab's scheduler in order to allow efficient, parallel execution. In particular, we split the scheduler into a single *global* scheduler (GS) and multiple *local* schedulers (LSs). The GS splits the work into large chunks, one for each local scheduler. The local scheduler manages the chunk and further partitions it into tasks. As we will see, these generalizations allow us to implement inference algorithms more efficiently.

In the rest of the section, we introduce the GIST API and its parallel execution model over a multi-core environment. We then discuss the implementation details of GIST in DataPath and PostgreSQL and discuss ways to implement GIST in other DBMSes.

3.1 GIST operator API

Like the treatment of UDA in DBMSes, GIST is an abstract interface that allows the system to execute GIST operators without knowledge of the specifics of the implementation. In this section, we present such an interface and refer to it as the *GIST API*. When designing the API, we have the following desirable properties in mind:

- *Do not restrict the state representation* Any such restriction limits the amount of *optimization* that can be performed. For example, GraphLab limits the state to generalized graphs, which deteriorate the performance. Graph-based state in Metropolis–Hastings algorithm in CDC forces the graph to be a fully connected graph. The CDC requires full consistency, but the full consistency locks all the nodes in the graph–which means no parallelization can be achieved. In the image denoising application, it achieves orders-of-magnitude speedup by using a matrix state instead of a graph state.
- *Allow fine grained control over the parallel execution* The applications are free to use their own synchronization primitives. Knowledge of the specifics of the problem allows selection of custom execution strategies. The problem may be better off to use lock-free schedulers and the best effort parallel execution [7,30], which relaxes the sequential consistency enforced in GraphLab to allow higher degree of parallelism.
- *Allow efficient mechanism to detect state convergence* Efficient mechanism is needed to detect the convergence in order to make termination decision. We design a convergence evaluation facility to gather statistics in parallel during task execution and make termination decision at the end of each round. This facility enables the computation of global statistics in parallel during inference. This mechanism is not supported in either MADlib or GraphLab.
- *Efficient system integration* Inference algorithms might require large initial states to be built, then post-processing and extraction of final results from such states. The GIST operator needs to *take over* states built by other means and allows efficient post-processing and extraction of the result. However, this type of mechanism is missing in GraphLab.

To achieve the above and allow a systematic GIST specification, all GIST operators are represented as a collection of five abstract data types: Task, Local Scheduler, Convergence UDA, GIST State and GIST Terminate. The GIST state will make use of the Task, Local Scheduler, Convergence UDA and GIST Terminate to provide a complete inference model. We discuss each part below starting with the sub-abstractions and finishing with the GIST State.

Task A task represents a single transition that needs to be made on the state and contains any information necessary to perform this transition. It may be a custom-made class, a class from a pre-made library, or even a basic C++ type. It is the job of the Local Scheduler to know what Tasks it needs to produce and the GIST to know what needs to be done to the state given a certain task. Essentially, the Task allows separation of planning and execution.

Local scheduler (LS) A LS is responsible for producing the Tasks used to perform state transitions. If the ordering of these Tasks is important, it is up to the LS to produce them in the correct order. Conceptually, the tasks specified by a LS are executed sequentially, but multiple LSs and their tasks may be executed in parallel. It is important to point out that the LSs do not execute tasks, they just specify which tasks should be executed. Effectively, the LSs contain part of the

execution plan to be used later by the GIST state. There is no requirement for the LSs to form a task sequence in advance—creating the task sequence on the fly is allowed. A LS has the following public interface:

```
class LocalScheduler {
      bool GetNextTask( Task& );
};
```

The `GetNextTask()` method stores the next task to be run in the location specified by the parameter. If there are no more tasks to be run, the method should return false, and true otherwise. In addition to this public interface, the LS can have one or more constructors that the GIST state is aware of.

Convergence UDA (cUDA) All inference algorithms need to detect convergence in order to make termination decisions. Detecting convergence requires statistics gathering followed by a termination/iteration decision. To allow such a mechanism to be specified, GIST requires a cUDA to be provided. A cUDA is a specialization of the UDA abstraction that is used to determine whether the GIST is done with inference. The cUDA is executed in parallel, much like a regular UDA. One cUDA instance is associated with one LS and gathers local statistics during the tasks execution for the corresponding LS through the use of `Accumulate`. At the end of a round, the cUDA instances are merged using `Merge` to obtain global statistics to allow the termination/iteration decision to be made through the method `ShouldIterate()`. Specifically, the cUDA API is:

```
class cUDA {
  void Init();
  void Accumulate(...);
  void Merge( cUDA& );
  bool ShouldIterate();
};
```

GIST state The GIST state represents the shared data used by threads of execution to perform the inference task. It contains all information that is global to all GIST functionalities and allows execution of the *abstract execution model (AEM)*. The AEM is a declarative execution model that allows the specification of parallelism without a commitment to a specific execution. First, the AEM of GIST specifies that the state transformation proceeds in *rounds*. Convergence is only checked on a round boundary, and thus inference can be stopped only at the end of a round. Second, the work to be performed in a round is split into many Tasks, which are grouped into bundles controlled by local schedulers (LSs). Tasks in a bundle are executed sequentially, but multiple threads of execution *can be* used to run on independent bundles. The LSs create/administer the Task bundles and provide the next task in a bundle, if any. The partitioning of a round's tasks into bundles is performed by the GIST State abstraction via the `PrepareRound` method and is, in fact, the planning phase of the round. This method plays the role of the *global scheduler*. This method should not perform any work—the user should assume that there is no parallelism during the execution of `PrepareRound`. The system provides a `numParts` hint that indicates a minimum number of LSs that should be created to take full advantage of the system's parallelization. To perform the work specified by each LS, the GIST State abstraction offers the method `DoStep`. This method executes the task provided as the input, i.e., one transformation of the state, and updates the statistics of the provided cUDA. At the end of a round, i.e., when all the tasks specified by all the LSs are executed, the system will merge the cUDA states and determine whether further iterations are needed. The execution proceeds to either another round or result extraction.

It is important to point out that this is just an abstract execution model. The execution engine will make the low-level decisions on how to break the work into actual parallel threads, how tasks are actually executed by each of the processors, how the work is split between GIST and other tasks that need to be performed, etc. The AEM allows enough flexibility for efficient computation while keeping the model simple.

The GIST constructor may either take constant literal arguments or pre-built states via state passing and prepares the initial state. Typically, a UDA is used to build the initial state in parallel, and to provide it in a convenient form to the GIST through the constructor.

The `PrepareRound` method produces the LSs and cUDAs for that round and places them into the vector provided. The integer parameter is a hint provided to the GIST for the number of work units that it can use. It is fine to provide more work units than the hint, but providing less will negatively impact the amount of parallelization that can be achieved. The `DoStep` method takes a Task and a cUDA and performs a single state transition using the information given by the Task. Any information related to the step's effect on the convergence of the GIST is fed to the cUDA.

```
class GIST {
  GIST(...);
  typedef pair<LocalScheduler*, cUDA*> WorkUnit;
  vector<WorkUnit> WorkUnitVector;
  void PrepareRound(WorkUnitVector&, int numParts);
  void DoStep(Task& task, cUDA& agg);
};
```

GIST terminate The Terminate facility allows for tuples to be post-processed and produced in parallel as long as the results can be broken into discrete fragments that have no effect on one another. The Terminate must have the following interfaces:

```
int GetNumFragments(void);
Iterator* Finalize(int);
bool GetNextResult(Iterator*, Attribute1Type&,...);
```

The GetNumFragments method is called first to return the number of fragments that the GIST can break the output into. The Finalize method takes as input the ID of the fragment to be produced and returns an iterator to keep track of tuples to be produced. The GetNextResult method uses the iterator and gets the actual tuples in location specified by parameter.

3.2 GIST execution model

Using the above API, the GIST can be executed as detailed in Algorithms 1 and 2. Algorithm 1 first initializes the GIST state (line 1) and organizes the work into rounds (lines 2–12). For each round, the local schedulers are produced into the list L (line 3) and then for each available CPU, an available local scheduler and cUDA are used to perform one unit of work (line 10). All the work proceeds in parallel until the unit of work terminates. In order to keep track of global work termination for the round, the workOut variable keeps track of the number of work units being worked on. When all work units are done and list L is empty, all the work in this round has finished. The variable gUDA is a global cUDA that is used to determine whether we need more rounds. Notice that round initialization and convergence detection are executed sequentially.

Algorithm 1: GIST Execution

```
Require: S_0 GIST Initial State, numCores, maxWork
 1: S← GIST State(S_0)
 2: repeat
 3:    L← S.PrepareRound(numCores)
 4:    workOut ← 0
 5:    gUDA ← Empty cUDA
 6:    repeat
 7:       C ← AvailableCPU
 8:       w ← Head(L)
 9:       workOut ← workOut+1
10:       C.PerformWork(w, maxWork, L, gUDA)
11:    until L.Empty() AND workOut == 0
12: until !gUDA.ShouldIterate()
```

Algorithm 2: PerformWork

```
Require: w WorkUnit, maxWork, L list<WorkUnit>, S GIST
   state, gUDA
 1: ticks ← 0
 2: repeat
 3:    t ← w.first.GetNextTask()
 4:    S.DoStep(t, w.second)
 5: until ticks>=maxWork OR t=empty
 6: if ticks=maxWork then
 7:    L.Insert(w);
 8: else
 9:    gUDA.Merge(w.second)
10: end if
11: workOut ← workOut-1
```

Parallelism of the GIST execution is ensured through parallel calls to PerformWork. As we can see from Algorithm 2, GetNextTask() is used on the local scheduler corresponding to this work unit to generate a task (line 3) and then the work specified by the task is actually executed (line 4). The process is repeated maxWork times or until all the work is performed (lines 2–5). Lines 6–10 detect whether we need more work on this work unit and whether convergence information needs to be incorporated into gUDA (this work unit is exhausted). The reason for the presence of maxWork is to allow the execution engine to *adaptively execute* the work required. If maxWork is selected such that the function PerformWork executes for 10–100 ms, there is no need to ensure *load balancing*. The system will adapt to changes in load. This is a technique used extensively in DataPath. In practice, if numCores argument of GIST Execution is selected to be 50% larger than the actual number of cores and maxWork is selected in around 1,000,000, the system will make use of all the available CPU with little scheduling overhead.

3.3 GIST implementation in datapath + GLADE

We implement GIST as part of the GLADE [39] framework built on top of DataPath [1] for cross-document coreference and image denoising. GLADE has a very advanced form of UDA called *Generalized Linear Aggregate* (GLA) that allows large internal state to be constructed in parallel and to be passed around to constructors of other abstractions like other GLAs or, in this case, GIST States. The above GIST execution model fits perfectly into DataPath's execution model. We add a waypoint (operator), GIST, to DataPath that implements the above execution model. The user specifies the specific GIST by providing a C++ source file implementing objects with the GIST API. The code generation facility in DataPath is used to generate the actual GIST Operator code around the user provided code. The planning part of the GIST Operator is executed by the DataPath execution engine in the same manner as the UDA, Join, Filter and other operators are executed. Through this integration, GIST operator makes use of the efficient data movement, fast I/O and multi-core execution of DataPath. Since the actual code executed is generated and compiled at runtime, the inference code encoded by GIST is as efficient as hand-written code.

In order to support the large states, which the GIST operator requires as input, we extended the GLA (DataPath+GLADE's UDA mechanism) to allow parallel construction of a single large state (to avoid costly merges) and *pass by STATE* mechanism to allow the state to be efficiently passed to the GIST operator. These modifications were relatively simple in DataPath due to the existing structure of the execution engine.

3.4 GIST implementation in PostgreSQL

We also implement GIST using MADlib [18] framework in Postgres for marginal inference over probabilistic knowledge graphs. The implementation follows the MADlib programming paradigm where the core of traditional SQL-SELECT... FROM... WHERE ...GROUP BY is used for orchestrating bulk data processing. After the data are prepared with the SQLs, it uses SQL/Python function to drive the algorithm. The SQL statements hide a lot of the complexity present in the actual operation making the network extraction extremely succinct. The database solution also enables us to do network extraction in a large dataset, which cannot fit into memory. The inner/core part of the machine learning/statistical methods are implemented using UDF/UDA with the C++ abstraction layer, which provides type bride to enable user to write vanilla C++ code. However for the state-parallel algorithms, where the state cannot be naively partitioned, the UDA is not a natural fit. We simulate the UDA–GIST interface to launch multiple threads in the finalize function. In order to enable user to run marginal inference query interactively on probabilistic knowledge graph in an RDBMS, we also implement a k-hop approximation which is elaborated in Sect. 6.

3.5 Discussion: implementing GIST in other DBMSes

In general, adding an operator to a DBMS is a non-trivial task. Both open source and commercial engines have significant complexity and diverse solutions to the design of the database engine. In general, there are two large classes of execution engines: MPP and shared-memory multi-threaded. We indicate how GIST can be incorporated into these two distinct types of engines.

MPP DBMSes, such as Greenplum, have a simple execution engine and use *communication* to transfer data between the execution engines. In general, the MPP engines are single-threaded and do not share memory/disk between instances. In such a database engine, it is hard to implement GIST since it requires the global state to be *shared* between all instances. On systems with many CPU cores and large amounts of memory in the master node, this obstacle could be circumvented by launching a program in the *Finalize* function of UDA which simulates the GIST. Such a program would have a dedicated execution engine that is multi-threaded and will perform its own scheduling. The *work_mem* parameter of the master node should be configured to be large enough to hold the entire state. Several limitations can be seen in this naive integration without DBMS source modification. Firstly, it cannot take advantage of other slave nodes in the MPP-GIST stage after the state is constructed in the UDA. Secondly, there is no support to convert the GIST state into DBMS relations in parallel. Lastly, converting from GIST internal state to a single value requires an extra copy of memory to store the state.

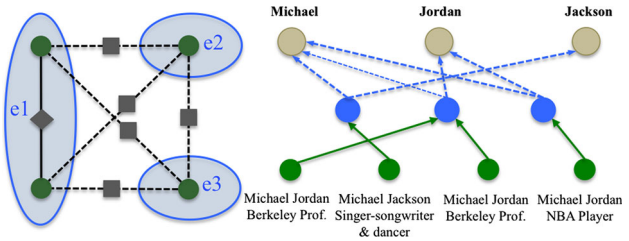


Fig. 3 GIST coreference pipeline. A UDA is used to construct the initial hierarchical model where each entity has one mention. The UDA passes the state into GIST, the GIST does parallel MCMC inference over the shared state until the state converges

Shared memory DBMSes, e.g., Oracle and DataPath, usually have sophisticated execution engines that manage many cores/disks. GIST, a shared memory operator, is a natural fit to the shared-memory DBMSes as evidenced by the deep integration of GIST to DataPath. To have a genuine integration to this type of DBMS, The UDA should be extended to support passing the constructed state into GIST. The GIST operator needs to make use of the low-level CPU scheduling facilities. Specifically how this can be accomplished depends on the specific DBMS implementation. As a practical integration, the source code of the integration of GIST to DataPath can be found in [14]. To have a shallow integration, the approach is similar as discussed in the integration of GIST to MPP DBMSes.

4 Application I: Cross-document coreference

In this section we first provide background knowledge of cross-document coreference [2] and the corresponding algorithms in Sect. 4.1. Then we discuss our implementation of those algorithm using UDA–GIST framework in detail in the following subsections.

Specifically for the implementation, the cross-document coreference process involves two distinct stages. The first stage builds the coreference initial state using a UDA, as illustrated in Sect. 4.2. Then a GIST parallel implementation of Metropolis–Hasting algorithm [10] is employed on the initial state until the state has been converged, which is explained in Sect. 4.3. Figure 3 depicts the implementation pipeline of cross-document coreference.

4.1 Metropolis–Hastings for cross-document coreference

Cross-document coreference [2] (CDC) is the process of grouping the mentions that refer to the same entity, into one

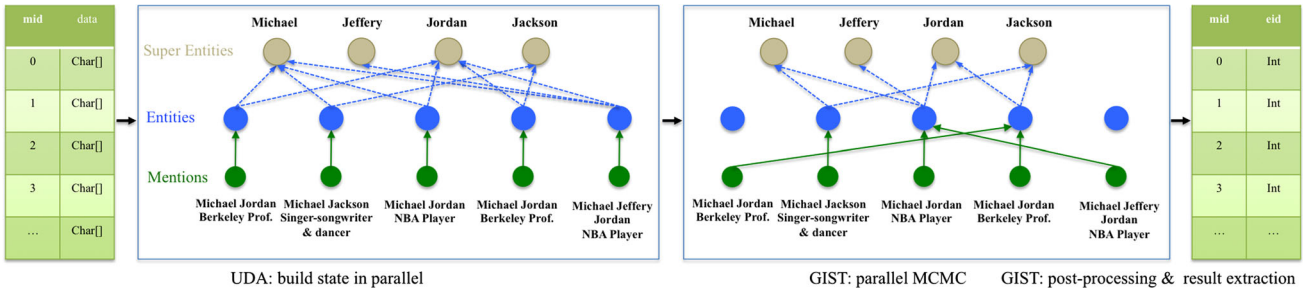


Fig. 4 Combined hierarchical model, a combination of pairwise model and hierarchical model. The *left graph* is a pairwise model consists of 4 mentions (*green circles*), 3 entities (*blue outlined ellipses*) and $C_4^2 = 6$ factors. One affinity factor is shown with *solid line*, and 5 repulsion factors with *dashed lines*. The *right graph* is the hierarchical model with extra two layers: the super entity and entity. The super entity is identified by a shared token in the top level. The entity represents a cluster of mentions that refer to the same real-world entity

entity cluster. Two mentions with different string literals can refer to the same entity. For example, "Addison's disease" and "chronic adrenal insufficiency" refer to the same disease entity. On the other hand, two mentions with the same string literal can refer to different entities. For example the name "Michael Jordan" can refer to different people entities. While CDC is a very important task, the state-of-the-art coreference model based on probabilistic graphical models is computationally intensive. A recent work by Google Research shows that such model can scale to 1.5 million mentions with hundreds of machines [44].

4.1.1 Pairwise factor model for coreference

A pairwise factor model is a state-of-the-art coreference model [44]. As shown in Fig. 4, the model consists of two types of random variables: entities (**E**) and mentions (**M**). Each mention can be assigned to one and only one entity, and each entity can have any number of mentions. There is one factor between any pair of mentions m_i and m_j. If the two mentions are in the same entity, the factor is an affinity factor $\psi_a(m_i, m_j)$; otherwise, the factor is a repulsive factor $\psi_r(m_i, m_j)$. Mathematically, we seek the maximum a posteriori (MAP) configuration:

$$\arg\max_{\mathbf{e}} p(\mathbf{e}) = \arg\max_{\mathbf{e}} \sum_{e \in \mathbf{e}} \Bigg\{ \sum_{m_i, m_j \in e, m_i \neq m_j} \psi_a(m_i, m_j) + \sum_{m_i \in e, m_j \notin e} \psi_r(m_i, m_j) \Bigg\} \quad (1)$$

Computing the exact **e** is intractable due to the large space of possible configuration. Instead, the state-of-the-art [44] uses Metropolis–Hastings sampling algorithm to compute the MAP configuration.

4.1.2 Hierarchical model for coreference

A hierarchical model is presented in a recent paper [44] to scale up CDC which improves the pairwise factor model using a two-level hierarchy of entities in addition to the base mentions: entities and super entities. Given the following concepts:

- $T(m)$: a set of tokens in the string literal of mention m.
- $T(e) = \cup_i T(m_i)$: a union of tokens in the set of mentions that belong to entity e.
- $P(e_s : m \rightarrow e_t, T(e_s) \cap T(e_t) \neq \emptyset)$: a proposal to move mention m from source e_s to destination e_t iff the two token sets $T(e_s)$ and $T(e_t)$ have at least one common token.

A super entity (**SE**) is a `map(key, value)`, where the `key` is a token t and the `value` is a set of entities, whose token set $T(e)$ contains token t. The SE is used as an index to quickly find the target entity in a proposal which has at least one common token with the source entity. The use of SE would increase the effectiveness of the sampling process P to achieve further scalability.

4.2 GIST building state in parallel

As explained in Sect. 2, the GIST operator relies on a UDA to construct the initial state with data in the mention relation. The UDA builds the initial state required for coreference by accumulating the super entity, entity and mention data structures, and the relationships among them. This is performed via the `Init`, `Accumulate` and `Merge` UDA functions:

Init Build containers for super entities, entities and mentions.

Accumulate The mention relation contains three columns: mention id, the mention string and the surrounding context of the mention. Each `Accumulate` call builds up one mention. Several steps are needed to process a mention. First, the mention string and the mention context are tokenized. Then all the stop words in the mention string are removed. Last, the mention tokens and context tokens are sorted to speed up

the calculation of cosine distance at the inference stage. Each `Accumulate` also builds one entity by taking the mention as input since each entity is initialized with one mention at the initial state (Fig. 3).

Merge `Merge` simply merges the lists of super entities, entities and mentions.

4.3 GIST parallel MCMC sampling

4.3.1 Technical issues

When implementing an application such as CDC using parallel MCMC sampling, a number of technical difficulties emerge. We briefly discuss one of them and our solution below.

Deadlock prevention The MCMC-based coreference algorithm we use needs to move mentions between entity clusters. This process involves inserting and removing mentions from entity data structures. Since these data structures are not atomic, race conditions can appear that can result in system crashes. The classic solution is to use locks to guard changes to the source and destination entities. Since two locks are needed, deadlocks are possible. To prevent this, we acquire the locks in a set order (based on entity ID).

4.3.2 GIST parallel MCMC implementation

The parallel MCMC can be expressed using the GIST with abstract data types: `Task`, `Scheduler`, `cUDA` and `GIST state`.

Task A task in the GIST implementation is a MCMC proposal which contains one source entity and one target entity. Furthermore, the `DoStep` function acquires the locks of the source entity and target entity in order. After obtaining the locks, it picks a random mention from the source entity, and then it proposes to move the source mention to the target entity. The task also keeps one reference to its `LS` since the task will consume some computation units, where the outstanding computation units are maintained in its `LS`.

Scheduler The global scheduler assigns the same amount of computation units, which is defined as one time factor computation between two mentions to each `LS`. The number of computation units is stored in variable `numPairs` in the LS. Each LS does Metropolis–Hastings inference until the computation units are consumed for the current iteration.

cUDA The cUDA is used to specify the MCMC inference stopping criteria. Usually, the convergence will be determined by a combination of the following criteria: (a) the maximum number of iterations is reached, (b) the sample acceptance ratio is below threshold, and (c) the difference of F1 between current iteration and last iteration is below threshold. The criterion of maximum number of iterations can be simply implemented by making sure the current iteration is not greater than the maximum number of iterations in the `ShouldIterate()`. To measure the sample acceptance ratio, each `cUDA` will measure the number of accepted proposals and the total number of proposals. Then all the cUDAs will be merged into one state to calculate the overall acceptance ratio. To use the third criterion, the `cUDA` needs to keep track of the last iteration F1 and current iteration F1 in the `cUDA`.

Algorithm 3: Metropolis–Hastings `DoStep`

```
Require: Task& task, LocalScheduler& ls
1: e_s ← rand(E)
2: m_s ← rand(e_s)
3: t = rand(T(e_s));
4: e_t ← rand(S(t))
5: lock e_s, e_t in order
6: if e^{p(e')/p(e)} > rand[0, 1] then
7:    accepted ← True
8: end if
9: if accepted then
10:   e_s ← e_s − m
11:   e_t ← e_t + m
12:   move the similar mentions in e_s to e_t
13: end if
14: unlock e_s, e_t in the reverse order;
```

GIST state The GIST state takes the state constructed in the UDA. In addition to the UDA state, GIST state defines the `DoStep` function that transforms the state by working on one task over the state. The `DoStep` function defined in the GIST state is described in Algorithm 3. Line 1 generates a random source entity from the entity space. Line 2 picks a random mention in the source entity. Line 3 produces a random token t in the token set of the source entity. Line 4 gets a random target entity in the super entity $S(t)$. Lines 5 and 14 acquire/release the locks of the source entity and target entity in order. Lines 6-13 perform one sample over the state. Line 12 moves all the similar mentions in the source entity to target entity.

5 Application II: Image denoising

In this section we first provide background knowledge of image denoising and the loopy belief propagation algorithm in 5.1. Then we discuss our implementation using UDA–GIST framework, depicted in Fig. 5. The implementation includes: (1) state building using UDA illustrated in Sect. 5.2; (2) GIST parallel LBP implementation in Sect. 5.3. LBP is

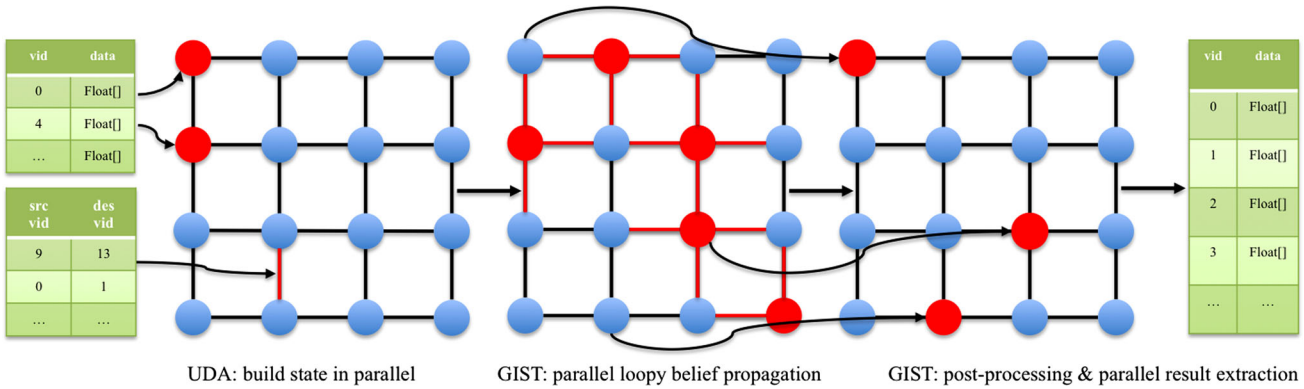


Fig. 5 GIST LBP pipeline. A UDA is used to construct the graph state. The UDA passes the state into GIST and the GIST does parallel loopy belief propagation over the shared state until the state converges, which is specified by the convergence UDA

one of the example application implemented in GraphLab. We show that our GIST API can be used to implement LBP and the UDA–GIST interface can be used to support the state construction, inference, post-processing and results extraction.

5.1 Loopy belief propagation for image denoising

The second SML application is image denoising, which is the process of removing noise from an image that is corrupted by additive white Gaussian noise. This is one of the example applications in GraphLab, which uses a 2D grid as the probabilistic graphical model, and 2D mixture as the edge potentials, which enforces neighboring pixels to have close color. The self-potentials are Gaussian centered around the observation. The output is the predicted image. Belief Propagation (BP) [16] is an inference algorithm on Bayesian networks and Markov random fields through message passing. Belief propagation operates on a bipartite factor graph containing nodes corresponding to variables V and factors U, with edges between variables and the factors in which they appear. We can write the joint mass function as:

$$p(\mathbf{x}) = \prod_{u \in U} f_u(\mathbf{x}_u) \tag{2}$$

where $\mathbf{x}_u$ is the vector of neighboring variable nodes to the factor node u. Any Bayesian network or Markov random field can be represented as a factor graph. The algorithm works by passing real valued functions called "messages" along the edges between the nodes. These contain the influence that one variable exerts on another.

A message from a variable node v to a factor node u is the product of the messages from all other neighboring factor nodes.

$$\mu_{v \to u}(x_v) = \prod_{u^* \in N(v) \setminus \{u\}} \mu_{u^* \to v}(x_v) \tag{3}$$

where $N(v)$ is the set of neighboring (factor) nodes to v. A message from a factor node u to a variable node v is the product of the factor with messages from all other nodes, marginalized over all variables except x and v:

$$\mu_{u \to v}(x_v) = \sum_{\mathbf{x}'_u : x'_v = x_v} f_u(\mathbf{x}'_u) \prod_{v^* \in N(u) \setminus \{v\}} \mu_{v^* \to u}(x_{v^*}). \tag{4}$$

BP was originally designed for acyclic graphical models. When BP is applied as an approximate inference algorithm over general graphical models, it is called loopy belief propagation (LBP), because graphs typically contain cycles. In practice, LBP converges in many practical applications [20].

5.2 Building GIST state in parallel

LBP is implemented in GraphLab using a generalized graph data structure since GraphLab uses a generalized graph as its underlying state, where any vertex can connect to any number of vertices either directed or undirected. In many cases, this is not the most efficient representation of the state space.

GIST API can efficiently support different data structures to represent the large state space. We implement this application model using two data structures: graph-based LBP and matrix-based LBP.

5.2.1 Graph-based GIST state

The GraphLab graph state representation is replicated in the GIST graph-based state, where the representation is achieved by maintaining a vector of all the vertices and a vector of all the edges. In addition, two vectors are maintained for each vertex: one vector to keep track of the edge IDs of the incoming edges and the other vector to keep track of the edge IDs of the outgoing edges.

Two UDAs are required to build the initial state. The `vertexUDA` builds up the graph nodes using the vertex

relation. The second UDA `edgeUDA` takes the `vertexUDA` and edge relation as inputs to produce the final graph state. For simplicity, we only shows the implementation of the `vertexUDA`.

Init In `Init`, the `vertexUDA` graph state is set up by allocating space for the vertex vector and two edge vectors for each vertex.

Accumulate Each tuple in the vertex relation is uniquely mapped to one element in the graph state. It achieves massive parallelism since the insertion of tuples into the graph state is done in parallel.

Merge The `Merge` function is left empty since the graph vertex state has been built in the `Accumulate` function.

5.2.2 Matrix-based GIST state

The above graph data structure is too general to exploit the structure in the problem: The pixel neighbors are always the up, down, left and right pixels in the image. A matrix representation captures this connectivity information in a very compact way—the matrix can be thought of as a highly specialized graph. Using a matrix data structure instead of a general graph data structure can significantly reduce the state building time since the state can be pre-allocated and the state initialization can be done in parallel. An `edgeUDA` to construct the edges is not needed in a matrix-based state since the edge connectivity is implicitly stored. Moreover, the performance of the inference algorithms developed on top of the data structure can be speeded up. In one run of the vertex program for one vertex, graph-based state requires a sequential access to the in-edge vector and out-edge vector to find the edge IDs and #|edges| random accesses to the global edge vector to modify the edge data.

In contrast, in a matrix state, it only requires one sequential scan to the edge data since the edge data for the vertex are maintained locally for each vertex. The detailed UDA implementation is described in the `Init`, `Accumulate` and `Merge` paragraphs.

Init In the `Init` function, a two dimensional matrix state for LBP inference is preallocated by taking the image dimensions.

Accumulate Each row in the vertex table stores a vertex id `vid` and the vertex's prior marginal probabilities. A vid uniquely maps to one cell in the matrix state. Thus, it is naively parallel since the initial data assignment for each element in state is independent.

Merge no code is specified in `Merge` since the state has been successfully built with only using the `Init` and `Accumulate`.

5.3 GIST parallel LBP implementation

The implementations of GIST parallel LBP based on the graph state and matrix state are almost identical with the help of GIST abstraction. For simplicity, only the matrix-state-based LBP implementation is described by demonstrating the implementation of the five abstract data types.

Task One task in the GIST LBP involves polling one vertex from the LocalScheduler (LS) queue and computing the belief based on the messages passed by its neighbors, then computing and sending new messages to its neighbors. Thus the *Task* contains one vertex id. It also keeps a reference to the LS since new task may be dynamically created and pushed back into the current LS.

Scheduler The global scheduler partitions the workloads into local schedulers evenly by assigning the same number of vertices into each LS. If one vertex needs to be resampled, a new task is generated and is pushed back to the end of the LS's task queue. There are no locks needed to extract tasks from LS since no two threads share a LS. In GraphLab, the new task might be inserted into another LS's task queue for the purpose of load balance. Although GraphLab has near-ideal load balance as shown in the experiment but at the cost of significant locking, which hampers the performance. GIST LBP implementation also relaxes sequential consistency enforced by GraphLab to allow higher degree of parallelism.

Convergence UDA The convergence criterion can be that the maximum number of iterations is reached or the residual of message value is below the terminate bound.

GIST state The GIST state takes the state constructed in the LBP UDA. In addition the UDA state, the GIST state defines the DoStep function that transforms the state by working on one task over the state. The DoStep function defined in the GIST state is described in the Algorithm 4. Lines 1 and 2 calculate the coordinates of the current vertex and the current vertex's neighbors. Lines 3–5 calculate the marginal probabilities (beliefs) based on the messages passed by its four neighbors. Lines 6–12 calculate the new messages to its neighbors. If the residual is above threshold, new tasks are dynamically created and added to the task queue. Please note that 'vid' refers to vertex ID, and 'dimen' is short for dimension.

GIST terminate After the LBP inference, the data for each vertex need posterior probability normalization. The frag-

Algorithm 4: LBP DoStep

Require: Task& task, LocalScheduler& *ls*
1: $V \leftarrow$ {task.vid%dimen, task.vid/dimen}
2: $N \leftarrow [top, bottom, left, right]$
3: **for all** $i \leftarrow 0$ **to** 4 **do**
4: update local belief $b(V)$ based on message $M_{N[i] \rightarrow V}$
5: **end for**
6: **for all** $i \leftarrow 0$ **to** 4 **do**
7: compute message $M_{V \rightarrow N[i]}$
8: $residual = ||M_{V \rightarrow N[i]} - M^{old}_{V \rightarrow N[i]}||$
9: **if** $residual > Terminate\ Bound$ **then**
10: $ls.addTask(N[i])$
11: **end if**
12: **end for**

ment output method allows for tuples to be post-processed and produced in parallel. This is an excellent choice for image denoising since it needs to produce large amounts of data as a result and the image can be broken into fragments that have no effect on one another.

6 Application III: Marginal inference queries over probabilistic knowledge graphs

In this section we first provide background knowledge of probabilistic knowledge graph and the marginal inference algorithms in Sect. 6.1. Then we introduce the k-hop approximation to further speed up the inference in Sect. 6.2. Finally we explain the marginal inference implementation, which involves two stages: the state building using a UDA explained in Sect. 6.3 and parallel GIST Gibbs sampling algorithm implementation in Sect. 6.4. Figure 6 illustrates the analytics pipeline of the in-database query-time marginal inference

6.1 Marginal inference queries over probabilistic knowledge graphs

Probabilistic knowledge bases (PKB), represented as Markov logic networks [37], are evolving by extracting new knowledge learned from rapidly growing amount of information on the web. Such information extraction systems learn over time to read the web and extract facts with confidence values.

The marginal probabilities of nodes in the factor graphs (the Markov network after grounding [37]) will change with the influence of new evidence learned from the web. Figure 7 shows a knowledge base of Barack Obama citizenship conspiracy theories constructed from a Wikipedia page of the events [53].

We see that in the ground predicate table, we have facts relevant to Obama's citizenship with prior extraction confidence. To determine probability of each fact with all correlated information, Formula 5 is used to calculate the exact marginal probability.

To answer a real-time marginal query in such a factor graph, we need to run inference over the cluster that the query node resides. Existing systems [9,35] are primarily designed for batch processing, not suited for interactive queries. Thus, we implement our k-hop approximation system in a RDBMS to support real-time queries. Users can issue a query to the RDBMS and obtain the marginal probability.

In the rest of this subsection we provide background knowledge of marginal inference over probabilistic knowledge graphs through Markov logic networks [37], factor graphs [41], and sampling algorithms such as Gibbs sampling [6] and MC-SAT [36] for marginal inference.

6.1.1 Markov logic network and factor graphs

Markov logic networks (MLNs) unify first-order logic [45] and probabilistic graphical models [23] into a single model. Essentially, a MLN is a set of weighted first-order formulae (F_i, W_i), the weights W_i indicating how likely the formula F_i is to be true. A simple example of a MLN is:

1. 0.96 born in (Ruth Gruber, New York)
2. 1.40 $\forall\ x \in$ Person, $\forall\ y \in$ City: born in$(x, y) \rightarrow$ live in(x, y)

It states a fact that Ruth Gruber is born in New York City and a rule that if a writer x is born in an area y, then x lives in y. However, both statements do not definitely hold. The weights 0.96 and 1.40 specify how strong they are; stronger rules are less likely to be violated.

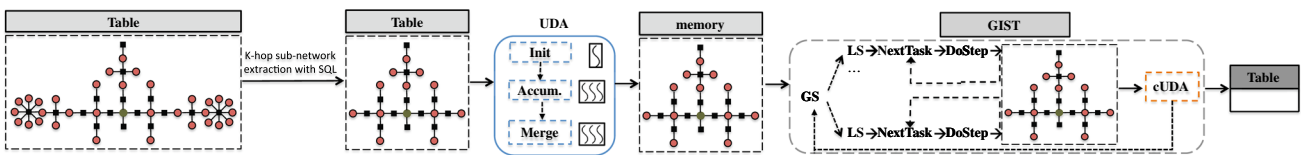
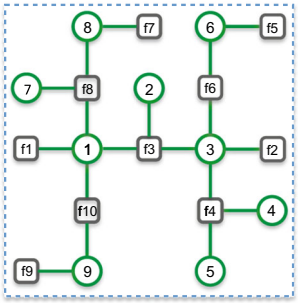


Fig. 6 The query-time inference analytics pipeline. First, k-hop sub-network centered with the query node is extracted using SQL with the user provided number of hops. then a UDA is used to load the k-hop network factors in table and glue the factors together into a single state. Lastly, in the finalized function, the Gibbs sampling is implemented using GIST interface. It returns the marginal probability of the query node to the user



(a)

factor id	clause	*prob*
f_1	1	1.00
f_2	3	1.00
f_3	$1 \wedge 2 \rightarrow 3$	1.00
f_4	$3 \wedge 4 \rightarrow 5$	0.95
f_5	6	0.30
f_6	$6 \rightarrow \neg 3$	1.00
f_7	8	0.95
f_8	$7 \wedge 8 \rightarrow 1$	1.00
f_9	9	0.95
f_{10}	$9 \rightarrow 1$	1.00

(b)

gpredicate id	ground predicate
1	Obama *isBornInState* Hawaii
2	Hawaii *isAStateOf* USA
3	Obama *isBornInCountry* USA
4	Obama *hasACitizenshipOf* USA
5	Obama *isEligibleToBePresidentOf* USA
6	Obama *isBornInCountry* Kenya
7	Star-Bulletin *isLocatedAt* Hawaii
8	Star-Bulletin *advertiseTheBirthOf* Obama
9	Obama *hasBirthCertificateIn* Hawaii

(c)

Fig. 7 Knowledge base of Barack Obama citizenship conspiracy theories. **a** Factor graph, **b** factor table, **c** ground predicate table

A MLN can be viewed as a template for constructing ground factor graphs. In the ground factor graph, each node represents a fact in the knowledge base, and each factor represents the causal relationship among the connected facts. For instance, suppose in the rule $\forall\, x \in$ Person, $\forall\, y \in$ City: born in$(x, y) \rightarrow$ live in(x, y), we have two nodes, one for the head "live in(x, y)" and the other for the body "born in(x, y)", and a factor connecting them, the values depending on the weight of the rule. The factors together determine a joint probability distribution over the facts in the KB.

A factor graph is a set of factors $\Phi = \{\phi_1, \ldots, \phi_N\}$, where each factor ϕ_i is a function over a random vector X_i indicating the causal relationships among the random variables in X_i. These factors together determine a joint probability distribution over the random vector X consisting of all the random variables in the factors. Mathematically, we seek the maximum a posteriori (MAP) configuration: it defines a probability distribution over its variables X:

$$P(\mathbf{X} = \mathbf{x}) = \frac{1}{Z} \prod_i \phi_i(\mathbf{x}) = \frac{1}{Z} \exp\Big(\sum_i w_i n_i(\mathbf{x})\Big) \tag{5}$$

where $n_i(x)$ is the number of true groundings of rule i in x, w_i is the rule weight, and Z is the normalization constant.

For more details about MLNs, we refer readers to [37].

6.1.2 Sampling for marginal inference

Computing the exact Z in Eq. 5 is intractable due to the large space of possible configuration. Sampling algorithms are typically used to approximate the marginal distribution since exact computation is difficult. Two most popular of these approaches are Gibbs sampling [6] and MC-SAT [36], described in the following two paragraphs.

Gibbs sampling Gibbs sampling [6] is a special case of the Metropolis–Hastings algorithm [10]. Gibbs Sampling can be used to approximate the marginal probabilities of each variable, where the joint distribution is unknown or hard to sample from, but the conditional probability of each variable is known and easier to sample from. The Gibbs Sampling algorithm is described in Algorithm 5:

Algorithm 5: Gibbs Sampling

1: $\mathbf{z}^{(0)} := \langle z_1^0, \ldots, z_k^0 \rangle$
2: **for** $t \leftarrow 1$ **to** T **do**
3: **for** $i \leftarrow 1$ **to** k **do**
4: $P(Z_i | z_1^{(t)}, \ldots, z_{i-1}^{(t)}, z_{i+1}^{(t-1)}, \ldots, z_k^{(t-1)})$
5: **end for**
6: **end for**

where T is number of iterations, and k is the number of variables.

The Gibbs sampling algorithm begins with random assigned initial value. Each variable is sampled from the distribution of that variable conditioned on all other variables, using the most recent values. The marginal probability of any variable can be approximated by averaging over all the samples of that variable. Usually, some number of samples (burn-in period) at the beginning are ignored, and then values of the left samples are averaged to compute the expectation. Gibbs sampling algorithms are implemented in the state-of-the-arts statistical relational learning and probabilistic logic inference software packages [22,35].

MC-SAT In real-world datasets, considerable numbers of Markov logic rules are deterministic. Deterministic dependencies break a probability distribution into disconnected region. When deterministic rules are present, Gibbs sampling tends to be trapped in a single region and never converges to the correct answers. MC-SAT [36] solves the problem by wrapping a procedure around the SampleSAT [49] uniform sampler that enables it to sample from highly non-uniform distributions over satisfying assignments, which is represented as $U_{SAT(M)}$ in Algorithm 6.

For more detailed discussion about MC-SAT, we refer to the original publication [36].

Algorithm 6: MC-SAT(*clauses*, *weight*, *num_samples*)

1: $\mathbf{z}^{(0)} \leftarrow$ Satisfy(hard clauses, i.e., clauses that are always true)
2: **for** $i \leftarrow 1$ **to** *num_samples* **do**
3: $M \leftarrow \emptyset$
4: **for** $c_k \in$ *clauses* satisfied by x^{i-1} **do**
5: With probability 1-e^{-w_k} add c_k to M
6: **end for**
7: Sample $x^{(i)} \sim U_{SAT(M)}$
8: **end for**

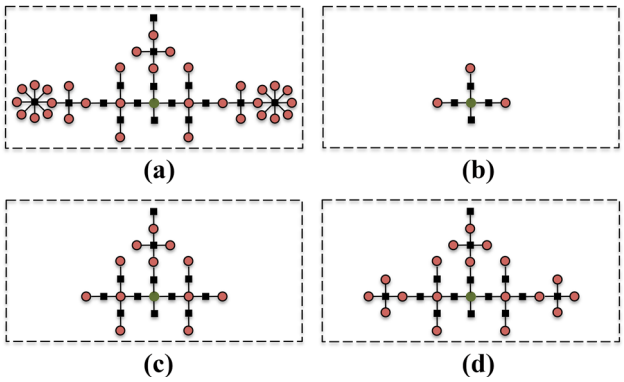


Fig. 8 k-hop network. **a** The original network and **b** 1-hop network. **c** 2-hop network and **d** 3-hop network

6.2 K-hop approximation

In this subsection we formally define the k-hop network, justify the k-hop approximation and explain the implementation. We also include a brief discussion on the extensibility of the k-hop approximation.

6.2.1 K-hop definition

In a factor graph, a factor is in the k-hop network if all of the literals in that factor is reachable in k hops from the query node. The following paragraph mathematically defines a k-hop network (Fig. 8).

Definition 1 Let the query node be a_q, the number of hops is k and the factor graph consists of N factors $\{C_i | 0 \leq i \leq N\}$. The factor $C_i = \{a_{i1} \vee a_{i2} \vee a_{i3} \cdots \vee a_{im}\}$ is in *k-hop network* iff for $\forall a_{ix}$ in the factor C_i $distance(a_q, a_{ix}) \leq k$, where the distance$(a_q, a_{ix})$ is the minimum number of hops between two nodes.

By specifying the k, users can directly control the size of the k-hop network. However, in some cases, it does not suffice to provide a k. There are cases that: (1) even with a small k, the k-hop network might be too large. For example, in a very a dense graph or even a complete graph, one k-hop network would cover most of the network; (2) The k-hop network is too small to have a good approximation, but $(k + 1)$ hop network is too large; (3) The k-hop network for query q_1 is small, but the k-hop network for query q_2 is too large. To make sure that the user can extract a relative big enough portion of the k-hop network in a query time, we set up the *node_limit* parameter that the k-hop network will return either the k hop is reached or the *node_limit* is reached.

6.2.2 K-hop justification

The k-hop network reduction is an effective method for reducing the search space for subsequent inference tasks. k-hop pre-processing can be reduced to efforts of focused belief propagation [8] and query aware MCMC [51]. Namely, the error incurred by not sampling certain nodes is proportional to the mutual information between it and the query node. We want to focus on influence on a small portion of the model early in the inference computation. Given a larger time budget, we should compute the full distribution; otherwise it is best to focus on *important* portions of the graph. The mutual information between two nodes is proportional to their active path. Therefore, as the number of hops between nodes increases, the effective error by not sampling a node decreases. When choosing a k we can set a threshold of the error we tolerate.

More formally, given a query node q, the importance of a node p is given by the following formula:

$$I_q(p) = \begin{cases} C & \text{if } \mathrm{dist}(p, q) = 1; \\ C/\mathrm{dist}^N & \text{if } \mathrm{dist}(p, q) > 1. \end{cases}$$

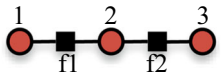
where C is some constant and N is a positive real value. Note that as the distance increases, the importance of the node diminishes.

We currently treat all facts as equal, but we can further prioritize nodes by using ad hoc pairwise similarity features. We can replace the C in the above algorithm with the pairwise influence score.

Extensibility of K-hop approximation The intuition behind k-hop approximation is straightforward: The farther away the node is from the query node, the less influence it has on the query node. Any application that involves a graph with such property and does not require exact result can use the k-hop approximation to avoid the cost of using the whole graph. The k-hop implementation is orthogonal to the actual algorithm implemented on top of the k-hop network, which means it can be simply added as an underlying layer of the actual algorithm implementation.

6.2.3 K-hop implementation

Specifically, the implementation includes three parts: (1) k-hop index construction; (2) k-hop network extraction; and



factors

fid	lits	weight
1	[-1,2]	0.9
2	[-2,3]	1.8

Fig. 9 An example factor graph

Table 1 K-hop index for the example factor graph

Factor_atom		Atom_factors		Edge	
Fid	Atom	Atom	Factors	SRC	Des
1	1	1	[1]	1	2
1	2	2	[1,2]	2	1
2	2	3	[2]	2	3
2	3	–	–	3	2

(3) k-hop index maintenance. The three parts of the implementation are discussed in the following parts.

Building K-hop index To speed up the k-hop network extraction, the k-hop index, which consists of three views `factor_atom`, `atom_factors` and `edge`, is built. The `factor_atom` builds the mapping from factors to atoms in that factor by unnesting the literal array to a set of rows. Each row stores the length of the literal array. For example, the factor with $fid = 1$ in Fig. 9 results in <1, 1, 2> and <1, 2, 2>. `atom_factors` builds the mapping from atom to an array of factors that the atom appears in. For example, atom 2 appears in the first and second factor, and thus [1, 2] is in the `factors` column. The `lens` column stores the length of the corresponding factor in the `factors` column. The view `edge` contains the reachability information. If two atoms appear in the same factor, then there are two entries stored in the `edge` since the factor graph is undirected. The k-hop index only needs to be built once as an initialization step for all queries (Table 1).

K-hop network retrieval Based on the definition in Section 6.2.1, to build the k-hop network, we need to find all the nodes that are reachable in k hops from the query node. This can be done via breadth first search (BFS) starting from the query node with a maximum of k level. Common Table Expressions (CTE) are used to do breadth first search and extract the k-hop network. The corresponding facility to CTE can be found in most of major RDBMSes such as Teradata, DB2, Microsoft SQL Server, Oracle, PostgreSQL, SQLite [52].

Incremental K-hop index maintenance To support the requirement of dynamic and evolving factor graph, two operators need to be supported in the RDBMS. These operations are **Insertion** and **Deletion**. Update operation is not discussed since it is equivalent to one deletion operation and one insertion operation. The k-hop index that contains the three views `atom_factors`, `factor_atom` and `edge` needs to be updated to reflect the changes to the factor graph. To update these views, the naive approach is to regenerate the k-hop index from scratch using the algorithms in Sect. 6.2.3. However, typically a incremental change is small, and an incremental maintenance strategy will be much faster than the naive approach, just by simply updating the affected tuples in the 3 k-hop index views. We implement SQL-based approach for updating the 3 k-hop index views by inserting/deleting relevant tuples in the 3 views for inserting/deletion of factors and report the performance in the experiment section.

6.3 State building with UDA

After the k-hop network is extracted, a UDA [12] is used to load and glue all the data together to form a single state. A UDA consists of three functions: Accumulate, Merge and Finalize.

Accumulate One tuple contains one factor as shown Fig. 9. The tuples are accumulated together in the *Accumulate* function of the UDA.

Merge The partial states in the UDA instances are merged into one single state. Single thread database such as PostgreSQL will skip the Merge function, and MPP databases such as Greenplum can merge the partial states in parallel.

Finalize SQL-based Gibbs sampling is very inefficient due to inefficient operations of random access to per-atom and per-clause with on-disk data [35]. However, these operations are the main operations of the Gibbs sampling. Instead, we choose to use RDBMS UDA to implement the Gibbs Sampling. The state in the Finalize function contains all the state information. The state passed into the GIST Gibbs sampler. The GIST Gibbs sampler does parallel Gibbs sampling and MC-SAT until the sampling algorithms converge. Finally, the result is written back to the database state and returned to the user.

6.4 GIST Parallel Gibbs sampling implementation

In the `Finalize` function of UDA, the Gibbs sampling and MC-SAT are implemented to do inference over the k-hop network. It is developed under the MADlib [19] framework and leverages the C++ abstraction layer that encapsulates DBMS-specifc logic.

In Algorithm 7, we show the Gibbs sampling for marginal inference. The implementation replicates the implementation in Alchemy [22], which includes a sequential in-memory

Algorithm 7: Sequential Gibbs Sampling

```
1: while done = false do
2:   for i ← 0 to numChains do
3:     performGibbsStep(i)
4:   end for
5:   if burnIn then
6:     burnIn ← checkConvergeAll(burnConvergeTests)
7:   else
8:     done ← checkConvergeAtLeast(convergeTests)
9:   end if
10: end while
```

implementation of Gibbs Sampling. In the implementation, we set the number of independent chains to 10 in line 2. It uses the burn-in criterion in Alchemy that throws out the samples at the beginning and continue sampling until 95% of the atoms converges according to the convergence criterion [22]. Lines 2–4 perform Gibbs sampling on 10 independent chains. Lines 5–10 detect the convergence of the burnIn of the Gibbs sampling.

The Algorithm 7 can be easily paralleled where multiple threads are launched and each thread runs one Gibbs chain. After each iteration, the results are combined together to evaluate the convergence of the Gibbs sampling.

Task One task in the GIST Gibbs involves sampling one Gibbs chain from the first node to the last node. The new values will be accumulated by the corresponding convergence UDA.

Scheduler The global scheduler partitions the workloads and passes them to local schedulers (LS) evenly by assigning equal number of Gibbs Sampling chains to each LS. Each LS does the work independently.

Convergence UDA The convergence criterion can be that the maximum number of iterations is reached or the number of converged atoms is above the termination threshold.

However, Gibbs sampling will not converge into the correct values with the presence of deterministic or near-deterministic rules, which are prevalent in real-world applications. Thus we implement MC-SAT [36], the state-of-the-art sampling algorithm for factor graph that can be invoked in the finalize function similar to the Gibbs Sampling implementation.

7 Experiments

The main goal of the experimental evaluation is to measure the performance of the UDA–GIST framework for the three problems exemplified in this paper: cross-document coreference (CDC), image denoising and query-time marginal inference. As we will see, when compared with the state-of-the-art, the GIST-based solutions scale to problems **27 times** larger for CDC and are up to **43 times** faster for image denoising. For query-time marginal inference, the k-hop approximation provides a trade-off between query time and accuracy. It returns high-quality result for marginal probability queries in a reasonable amount of time. Also our parallel Gibbs sampling achieves one order of magnitude speedup. We also include evaluation with MC-SAT and k-hop approximation to support deterministic rules efficiently.

7.1 Experimental setup

7.1.1 Cross-document coreference and image denoising

In order to evaluate the C++ GIST implementation of CDC and image denoising, we conduct experiments on various datasets in a multi-core machine with 4 AMD Opteron 6168 running at 1.9GHz processors, 48-core, 256GB of RAM and 76 hard drives connected through 3 RAID controllers. The UDA–GIST framework is implemented in DataPath [1], an open source column-oriented DBMS. The implementation makes use of the GLA, the UDA facility in GLADE [39]. For the image denoising application, we compare the performance with GraphLab. Although we are not able to replicate the same experiment for the CDC in [44] since its source code is not published and the running time is not reported, which makes the direct comparison impossible, we show that we are able to tackle a 27 times larger problem than the problem solved in [44]. To the best of our knowledge, GraphLab and Google CDC are written in C++.

7.1.2 Marginal inference queries over probabilistic knowledge graphs

In order to evaluate runtime and accuracy of the in-RDBMS inference with the k-hop approximation, we conduct experiments on a PostgreSQL DB over a 32-core machine with 2T hard drive and 64 GB memory. We use the ReVerb-Sherlock dataset and NELL- Sport dataset as discussed in Sect. 7.4.1. We evaluate the accuracy the of k-hop approach against with the ground truth on the entire factor graph.

7.2 Cross-document coreference with Metropolis–Hastings

7.2.1 Cross-document coreference datasets

The performance and scalability of GIST implementation of cross-document coreference (CDC) is evaluated using the Wikilinks [43] dataset. The Wikilinks dataset contains about 40 millions mentions over 3 millions entities. The surrounding context of mentions is extracted and used as context

features in the CDC techniques. In this dataset, each mention has at most 25 left tokens and at most 25 right tokens. The dataset is generated and labeled from the hyperlinks to the Wikipedia page. The anchor texts are *mentions*, and the corresponding Wikipedia hyperlinks are *entities*. Any anchor texts which link to the same Wikipedia page refer to the same entity. For our experiments, we extract two datasets Wikilink 1.5 (first 1.5 M mentions from the 40 M dataset) and Wikilink 40 (all 40 M mentions in the dataset) from this Wikilink dataset.

The state-of-the-art [44] evaluates CDC performance using a different Wikilink dataset containing 1.5 million mentions. The exact dataset used and the running time in that paper are not published, and thus a direct comparison is not possible. Nevertheless, our version of the Wikilink 1.5 has the same number of mentions and about the same number of entities.

Notably, with the exception of [44], no prior work provided experiments with datasets larger than 60,000 mentions and 4000 entities (see [43] for a discussion). The Wikilink 40 dataset is **27 times** larger than the largest experiment reported in the literature.

7.2.2 Experiment result

Methods The performance of GIST coreference is evaluated against the state-of-the-art [44] using a similar dataset, feature set, model and inference method.

- Datasets: Wikilink 1.5 and Wikilink 40
- Feature set: the same feature set as in [44] are used where the similarity of two mentions m and n is defined as:

$$\psi(m, n) = (\cos(m, n) + wTSET\_EQ(m, n))$$

Where $cos(m, n)$ is the cosine distance of the context between mention m and mention n. $TSET\_EQ(m, n)$ is 1 if mentions m and n have the same bag of words disregarding the word order in the mention string, otherwise it returns 0. w is the weight of this feature. In our experiment, we set $w = 0.8$.

To evaluate the accuracy of CDC results, we use precision and recall, as well as F_1 score, the harmonic mean of precision and recall.

Wikilink 1.5 experimental results The performance evaluation results over Wikilink 1.5 are depicted in Fig. 10a. The experiment runs for 20 iterations. Each iteration takes approximately 1 min. During initialization (iteration 0), each entity is assigned to exactly one mention. The state is built using a UDA in about 10s. The inference starts at iteration 1

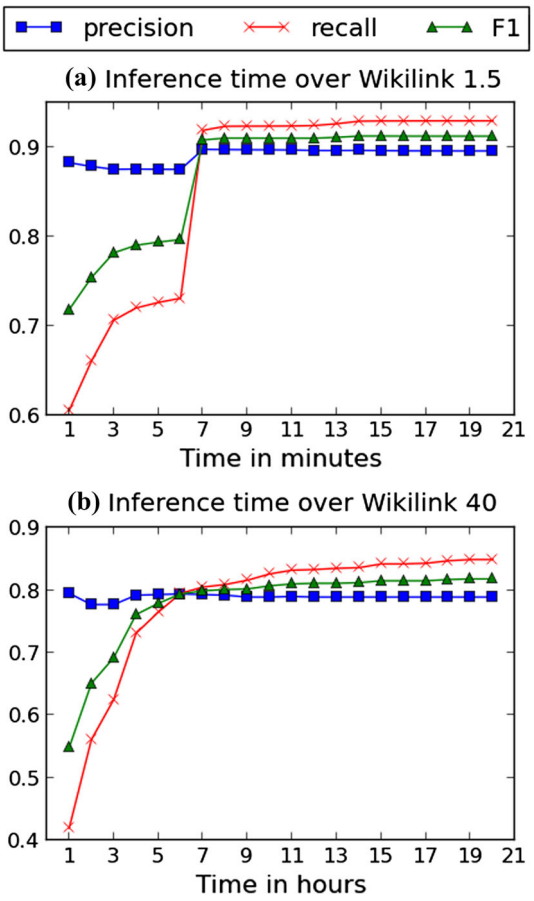


Fig. 10 GIST MCMC evaluation over Wikilink 1.5 and Wikilink 40

after the state is constructed. At iteration 7, the state essentially converges and has precision 0.898, recall 0.919 and F_1 0.907. The F_1 continues to slightly improve up to iteration 20. In the last iteration 20, the measures are precision 0.896, recall 0.929 and F_1 0.912. [44] employs 100–500 machines to inference over a similar dataset.

Wikilink 40 experimental results To evaluate the scalability of our coreference implementation, we use the Wikilink 40 that is 27 times larger than the current state-of-the-art. As the same as the above experiment, each entity is assigned with exactly one mention during initialization. The state building takes approximately 10 min. Figure 10b depicts the performance of our implementation with this dataset on the experimental machine. We run 20 iterations each with 10^{11} pairwise mention comparisons—each `LocalScheduler` is generating random tasks until the comparisons quota is met. Each iteration takes approximately 1 h, and we can see that the graph converges at iteration 10 with precision 0.79, recall 0.83 and F_1 0.81. This essentially means that, using our solution, within a manageable 10 h computation in a single system the coreference analysis can be performed on the entire Wikilink dataset (Fig. 10).

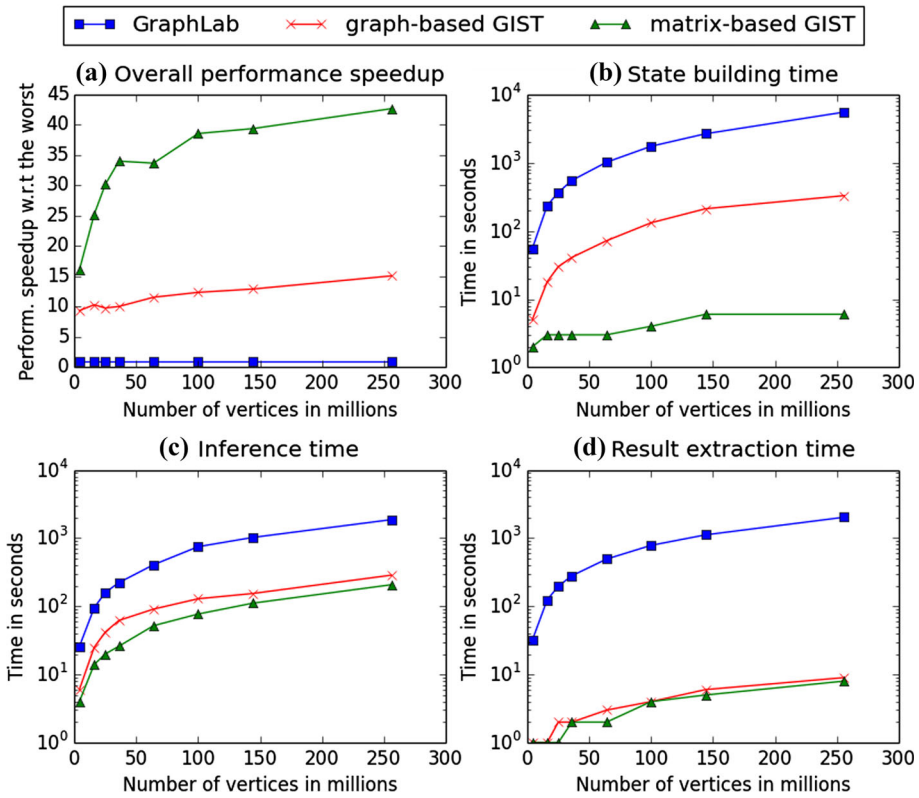


Fig. 11 Matrix-based, graph-based GIST LBP (best effort) and GraphLab LBP (sync. engine) performance evaluation over the image datasets

Discussion A direct comparison to the state-of-the-art [44] is not possible since the dataset used is not published and the time plot in its performance graph is relative time. We believe our results are substantial since our final inference measure is similar as reported in the paper and our experiment evaluation finishes in 10 min on a single multi-core machine instead of 100–500 machines for a similar dataset Wikilink 1.5. We are also able to finish the inference in 10 h for a 27 times larger dataset Wikilink 40. The speedup can be seen as follows: [44] uses MapReduce to distribute the entities among machines. After sampling the subgraphs, the subgraphs need to be shuffled and even reconstructed between machines, which suffers I/O bottleneck. However, we use one single machine with enough memory to store the whole graph and maintain an in-memory super entity structure to speed up the MCMC sampling.

7.3 Image denoising with loopy belief propagation

7.3.1 Image denoising datasets

We evaluate the performance with synthetic data generated by a synthetic dataset generator provided by GraphLab [27]. The generator produces a noisy image Fig. 12b and the corresponding original image Fig. 12a. Loopy belief propagation is applied to reconstruct the image, and it produces a predicted image. We use the dataset generator to generate 10 image datasets varying from 4 millions pixels (2000×2000) to 625 million pixels ($25{,}000 \times 25{,}000$).

7.3.2 Experiment result

We evaluate the performance of the three approaches: GraphLab LBP, GIST matrix-based LBP and GIST graph-based LBP against the 10 image datasets. The analytical pipeline consists of three stages: state building, inference and result extraction. Figure 11 provides a detailed performance comparison of the three methods for each of the three stages. Due to the more compact representation (no explicit representation of edges), only the matrix-based GIST implementation can build the state with 400 millions and 625 millions vertices image dataset. We are able to perform experiments on images with 4–256 million pixels for all three methods.

Overall performance comparison To sum up all the performance metrics of the three stages, Fig. 11a describes the overall performance speedup w.r.t. the worst. Clearly, GraphLab performs the worst and its performance speedup against itself is always 1. As shown in the experiment results, graph-based GIST can achieve up to 15 times speedup compared with GraphLab. With a matrix-based abstraction, GIST can achieve up to 43 times speedup compared with GraphLab.

State building In the state building phase, as we can see from Fig. 11b, the graph-based GIST outperforms the GraphLab by up to 16 times speedup with a UDA to construct the graph state in parallel. It is mainly due to the parallel I/O in the DBMS where each UDA instance loads one chunk of the vertex and edge data into memory and the final graph is merged together in the merge function of UDA. GraphLab sequentially constructs the graph state without parallelism as suggested in Fig. 13a, where only one CPU core is used. Matrix-based GIST further improves the state building using a matrix instead of a general graph as the underlying state. The time to build the graph state using a matrix-based GIST is three orders of magnitude faster than the GraphLab. In the matrix-based GIST, a matrix is pre-allocated and UDA instances can pull the data from the disk and fill the matrix independently with massive parallelism.

LBP inference With the identical algorithm implemented in Graph-Lab, GIST LBP produces the same quality image as GraphLab as shown in Fig. 12. The number of vertices sampled in each of the settings is in range [1.45 billion, 1.48 billion].

GraphLab with sync. engine, with async. engine, sweep scheduler and with asyn. engine, fifo_queue scheduler takes about 30, 26, 24 m to converge, respectively. Graph-based GIST LBP only takes 4.3 m with the lock-free scheduler as discussed in Sect. 5.3. Matrix-based GIST LBP further improves the running time to 3.2 m. The 27% performance difference between graph-based and matrix-based GIST is due to the better memory access pattern of the matrix. GraphLab's CPU utilization with sync. engine fluctuates between 6.0 and 45.5 out of 48, where GIST can almost fully utilizes the 48 cores. GraphLab enforces load balancing using the two-choices algorithm in [32] through locking two random task queues. The load balance is not an issue as indicated by the steep decline curve at the end of inference, but the cost of locking is significant since the tasks are very lightweight (involving 4 neighbors). Considering data race is rare in a graph with hundreds of millions of nodes for 48 threads, GIST LBP further relaxes sequential consistency to allow high degree of parallelism. Matrix-based GIST LBP with best effort parallel execution (relaxing sequential consistency) converges to the correct point, shown in Fig. 12, and improves the running time to 2.5 m.

GIST terminate After the inference, the posterior probability values in vertices of the graph need to be normalized, and then results need to be extracted. GraphLab does not support post-processing and results extraction in parallel since it only has an abstraction for inference. After the parallel inference, GraphLab post-processes each vertex sequentially as shown in the timeline [122, 160] minute of Fig. 13a. The GIST Terminate facility allows for multiple tuples to be post-processed and produced in parallel as depicted in the timeline [3.78, 3.92] minute of Fig. 13b, and thus it achieves more than two-orders-of-magnitude speedup over GraphLab.

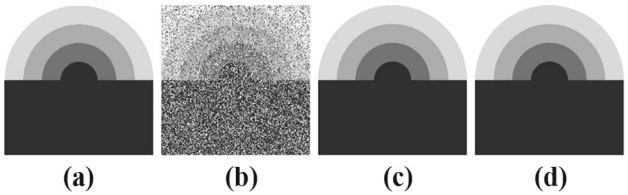


Fig. 12 Image denoising with LBP. **a** The original image and **b** is the corrupted image. **c** The predicted image by GraphLab and **d** is the predicted image by GIST

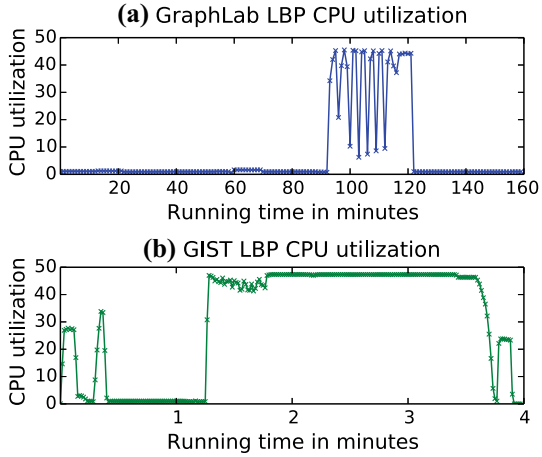


Fig. 13 GraphLab LBP (sync. engine) and GIST matrix-based LBP (best effort) CPU utilization evaluation with the image with 256 millions pixels in a single machine with 48 cores

7.4 Marginal inference queries with K-hop approximation over probabilistic knowledge graphs

7.4.1 Query-time marginal inference queries dataset

In our experiment, we evaluate our approach over two real-world datasets: ReVerb knowledge base [15] using the Sherlock inference rules in [40] and NELL candidate belief dataset [31] using inference rules in [24]. These rules are uncertain, forming a Markov logic network (MLN) [37]. The MLN inference involves two steps:

1) Generating a ground factor graph, described in [9];
2) K-hop inference over the ground factor graph.

ReVerb-Sherlock The original ReVerb-Sherlock KB contains 407,247 facts and 30,912 inference rules. We run the grounding algorithm described in [9] and generate a factor graph with 54,103 nodes and half a million factors. Notably, we normalize the weight in the dataset to double value in

Table 2 Factor graph cluster size distribution

Cluster size	#Clusters	Percent
(a) ReVerb-Sherlock		
453,384	1	90.7
[1056, 3649]	4	1.4
[1, 711]	3578	7.9%
(b) NELL- Sport		
714,760	1	19.5
[10335, 141811]	65	46.9
[3, 9988]	1518	33.6

[0,1] since large weight would make the Gibbs sampling hard to converge in a finite amount of time as discussed in [36]. However our focus is on evaluating the effectiveness of k-hop approach instead of the underlying inference algorithms and the k-hop approach is orthogonal with the effort to improving the inference algorithm. Our experiment performs k-hop inference over this factor graph and aims at determining credibilities of the queried facts based on the input KB and rules. Table 2 shows the component size distribution in the factor graph. It shows 90% of the clauses are in the biggest cluster. We categorize this cluster as large cluster. Similarly, we categorize the cluster with 3649 clauses as the medium cluster and the cluster with 711 clauses as the small cluster.

NELL- Sport The original NELL candidate belief dataset contains 84.6 million facts and 1828 rules in the sport domain. Some rules with same rule type but have different weights. We remove the duplicate rules and only keep one single unique rule with the averaged weight of duplicate rules. All the rules that contains "generalize" predicate are removed since they do not relate in the sports domain. We run the grounding algorithm [9] which takes around 25 min. Finally, we get a factor graph with 23,3756 facts and 3,670,822 factors.

7.4.2 Experiment result

This part shows performance of marginal inference queries with k-hop approximation in PostgreSQL in terms of runtime and accuracy.

K-hop index maintenance evaluation As described in Sect. 6.2.3, the k-hop index needs to be built before any k-hop query can be issued. It takes about 35.1 s to build the k-hop index for the ReVerb- Sherlock KB and 132.6 s for the NELL- Sport KB. With incremental changes to the factor graphs, the indexes need to be maintained properly to reflect the changes in the factor graphs. Table 3 describes the k-hop index maintenance cost of insertion and deletion with various incremental changes from 0.001 to 10%.

Table 3 Incremental k-hop index maintenance cost

Percent	Delete(s)	Insert (s)
(a) ReVerb-Sherlock		
0.01	1.4	0.65
0.1	10	2.45
1	22.5	5.9
2	23.9	6.9
4	27.2	9.1
6	28.9	11.3
8	31.3	12.8
10	36.1	16.5
(b) NELL- Sport		
0.01	15.2	6.9
0.1	51.7	30.2
1	83.1	45.4
2	86.5	52.8
4	98.4	66.1
6	109.7	78.9
8	115.3	90.8
10	119.2	100.8

To evaluate the incremental maintenance cost after deletion, we randomly delete factors from 0.001 to 10% of the original factor graph. As depicted in Table 3, it takes only 1.4 and 15.2 s to maintain the indexes for the ReVerb-Sherlock and the NELL- Sport after deleting 0.01% of the original factor graphs. The maintenance cost increases as the deletion increases. With the 10% deletion in the ReVerb-Sherlock, it takes 36.1 s to maintenance the index view. It suggests the naive approach of regenerating these views would be faster if more than 10% factors need to be deleted. However in NELL- Sport, it takes 119.2 s to maintain the index with 10% deletion which is marginally better than the naive approach that requires 132.6 s.

For incremental maintenance cost after insertion, we randomly select 90% of the original factor graphs as the base factor graphs for the two datasets and take the un-selected factors as the insertion. The result is similar to incremental deletion, but even faster.

In sum, a typical update to the KB is usually small, even less than 0.01%. Our results show that it achieves an order of magnitude speedup than the naive approach for 0.01% deletion and 0.01% insertion. It is still marginally better or almost equivalent than the naive approach with 10% update in the two factor graphs.

K-hop parameters setup As discussed in Sect. 6.2.1, the two parameters `num_hop` and `node_limit` control the k-hop network size, thus affecting the approximation accuracy and runtime. We evaluate the effect of the first parame-

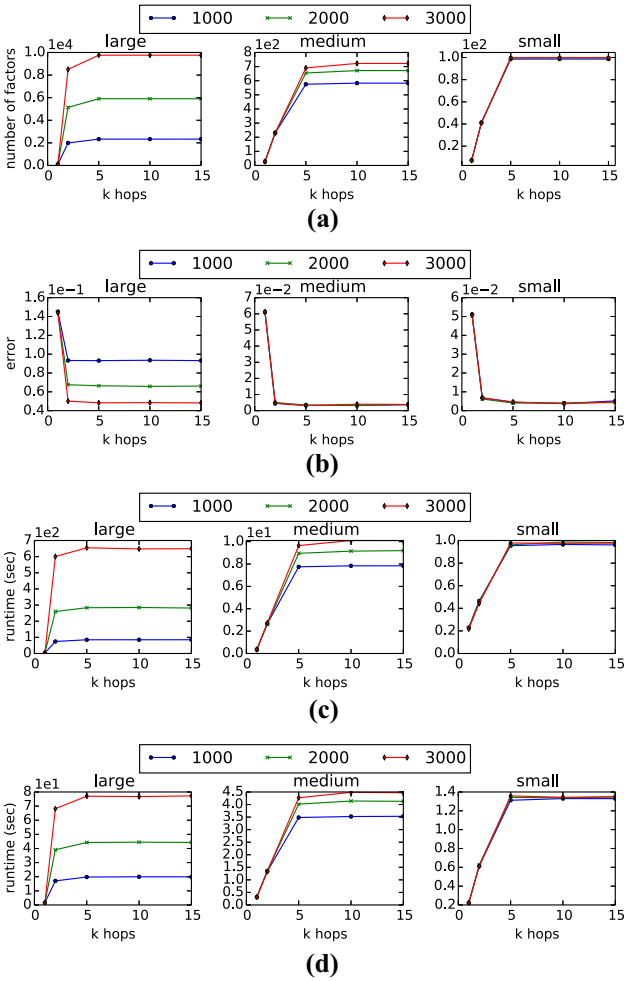


Fig. 14 REVERB- SHERLOCK k-hop evaluation with Gibbs Sampling. **a** Network size, **b** error, **c** runtime with sequential execution, **d** runtime with parallel execution

ter by varying `num_hop` from 1 to 15 and the effect of the second parameter by setting `node_limit` with three different values: 1000, 2000 and 3000. We compare the performance of parallel Gibbs sampling and sequential Gibbs sampling. Moreover, since the size of the cluster where the query node resides has a direct effect on the performance of inference using k-hop approximation, we categorize the clusters into three types: large, medium and small, as discussed in Sect. 7.4.1 and we select one cluster from each of the three types with the largest size in that type of cluster. It is impractical to evaluate all the nodes in the factor graph cluster since it can be time-consuming and unnecessary. Thus we randomly sample 50 nodes from three clusters and use the average results instead.

Evaluation over REVERB- SHERLOCK. Figure 14 shows the evaluation results on REVERB- SHERLOCK with Gibbs Sampling in terms of k-hop network size, error, runtime with different settings of `num_hop` and `node_limit`. The graphs in left, middle and right in Fig. 14 describe the average results of the 50 nodes from the large, medium and the small clusters, respectively. Figure 14a describes the k-hop network sizes with different settings of parameters. In the large cluster, we see that the larger the limit, the larger the retrieved k-hop network. Also the network size increases with the increase of the number of hops until `num_hop` reaches 5. Thus it suggests the `node_limit` is reached before the `num_hop` is reached. In the medium cluster, the difference of `node_limit` values is not as significant as in the large cluster. In the small cluster, there is almost no difference as we see that the lines of the various `node_limits` almost overlap, indicating that `num_hop` is reached first before `node_limit` is reached. After 5 hops, it returns the whole factor graph for a small factor graph. Figure 14b shows the error. As shown in Fig. 14b, the error reduces from 0.145 to 0.11 with `node_limit` = 1000 as we increase hop from 1 to 5. With `node_limit` = 3000, the error is further reduced to 0.06. For medium and small clusters, the error is reduced from hop 1 to hop 2. With larger hops, the error becomes negligible. Figure 14c shows the performance of sequential execution and parallel execution. It is consistent with Fig. 14a since the neighbor network size decides the runtime. The parallel implementation achieves one order of magnitude speedup compared to the sequential implementation. With smaller networks, the improvement is not significant since the k-hop network extraction would be a dominant factor of the overall runtime.

Evaluation over NELL- SPORT Figure 15 shows the evaluation on NELL- SPORT with MC-SAT, since Gibbs sampling will not converge to the correct values when deterministic and near-deterministic rules are present in the NELL-Sport dataset. Due to the single-chain nature of the MC-SAT algorithm, it cannot be implemented with GIST parallel interface efficiently, and thus we only provide the sequential runtime to demonstrate the k-hop effectiveness of runtime/accuracy trade-off. The results are similar to REVERB- SHERLOCK, except that the three lines with different `node_limit` coincide, due to more evenly distributed nodes in different sizes of clusters than REVERB- SHERLOCK.

8 Related work

Several significant attempts have been made toward efficient computation frameworks for SML in DBMSes such as MADlib [11,18] and in other parallel and distributed frameworks such as MapReduce [13,29], GraphLab [27,28] and GraphX [54].

MADlib [11,18] integrates data-parallel SML algorithms into DBMSes. By allowing a Python driver for iterations and a UDA to parallelize the computation within each iteration, algorithms like logistic regression, CRF and K-means

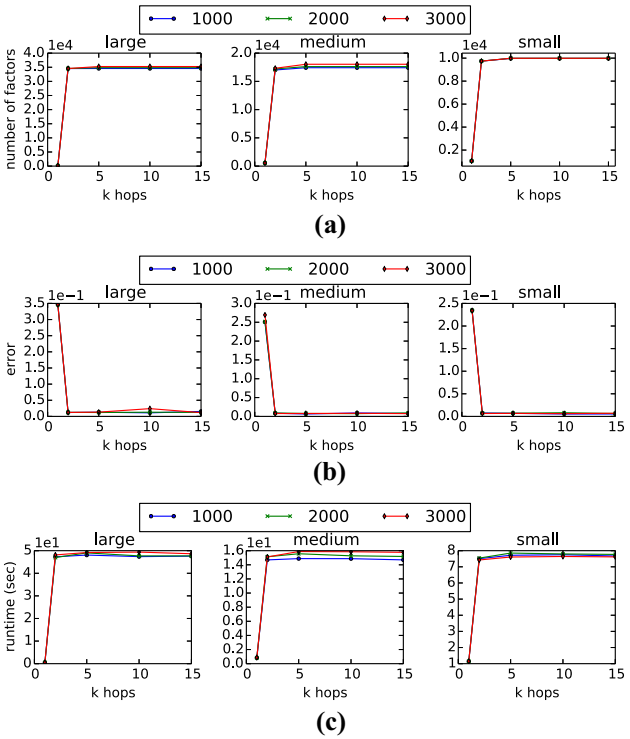


Fig. 15 NELL- SPORT k-hop evaluation with MC-SAT. **a** Network size, **b** error, **c** runtime with sequential execution

algorithms are implemented efficiently [25]. However, the data-driven operator, UDA, cannot efficiently express state-parallel algorithms.

Tuffy [35] attempts an in-database implementation of WalkSAT algorithm over Markov Logic Networks (MLN), but it is too slow for practical use. Tuffy results in a hybrid architecture where grounding is performed in DBMS and WalkSAT is performed outside of the DBMS. The grounding step over MLN joins the first-order model with data to construct the model, which is the state space consisting of nodes and edges. The sampling step over the MLN is performed outside of a DBMS due to the inefficiency of state-parallel algorithm using the data-driven execution model. Similar to UDAs, MapReduce excels at expressing data-parallel algorithms, but it cannot efficiently support state-parallel SML algorithms.

To address the limitations of data-parallel operators to express graph-parallel SML algorithms, GraphLab proposes a computation framework with a graph-based abstraction for graph-parallel algorithms. GraphLab simplifies the design and implementation of SML algorithms , but it cannot express SML algorithms whose underlying states are complete graphs, dynamic graphs or more general data structures. As a result, the CDC using the Metropolis–Hastings algorithm where the underlying state is a complete graph cannot be implemented efficiently. Secondly, GraphLab misses the opportunity to exploit the structure of specific problems. For example, the state in the image denoising application can be represented as matrix, a specialized graph. A matrix state brings the opportunity to build the state in parallel with a UDA. It also speeds up the inference due to the better access pattern of matrix. Compared to GraphLab, the UDA–GIST framework further speeds up the performance using lock-free schedulers and best effort parallel execution, which relaxes sequential consistency to allow higher degree of parallelism [7,30]. Moreover, GraphLab is not integrated with a scalable data processing systems for parallel state construction and parallel result extraction. It is difficult for GraphLab to connect to a DBMS to support a query-driven interface over the data and result due to the impedance mismatch of non-relational world and relational engine.

Pre-processing and post-processing in a graph analytical pipeline are time-consuming, which even exceeds the inference time. Motivated by that, GraphX, built on Spark [55], inherits the built-in data-parallel operator to speed up the pre-processing and post-processing. It produces triplets table to represent a graph by joining vertex relation and edge relation. In essence, GraphX is in the same spirit as MapReduce since it is based on a synchronous engine and has data duplication to represent a graph which is different from the graph representation in GraphLab. For inference, GraphX is less efficient than the GraphLab, but it outperforms GraphLab from a end-to-end benchmark, which consists of pre-processing, inference and post-processing [54].

Researchers have created systems for large-scale knowledge base construction and inference including Alchemy [22], Tuffy [35], NELL [5] and ProbKB [9]. Alchemy provides a series of algorithms for in-memory statistical relational learning and probabilistic logic inference for Markov logic network. Tuffy improves the grounding phase of inference using RDBMS. However, there is no parallelization in both Alchemy and Tuffy for inference in a MLN network. ProbKB achieves significant speedup in the first phase of grounding using MPP Databases Greenplum.

To support query-time inference queries over probabilistic graphical models, Sümer et al. [46] present adaptive inference in graphical models by taking advantage of previously computed quantities to perform inference more rapidly than from scratch. However, this approach can only be used in exact inference and small factor graphs. Jiang et al. [21] present a similar idea to our approach but limit the neighbor network to 2 hops. Also the definition of the 2 hops is not formally defined.

There are also other works aiming on scalable inference using graphical models. Niepert et al. [34] use Tractable Markov Logic (TML), a subset of Markov Logic to achieve tractable inference, while limiting the expressiveness only comparable to probabilistic Horn KBs. Wick et al. [50] use MCMC and factor graph for scalable query over probabilistic

database, which does not support for query-time response. Beedkar et al. [4] achieve parallel inference via partitioning the Markov logic network and importance sampling; however, the method is much more complicated than our GIST implementation while achieving modest speedup. Shin et al. [42] focus on statistical inference on the updated factor graph with incremental changes, while our work focuses on approximate inference in real time. However, those works aforementioned do not support query-time inference.

9 Conclusion

In this paper we introduce the GIST operator to implement state-parallel SML algorithms. We present the UDA–GIST, an in-database framework, to unify data-parallel and state-parallel analytics in a single system with a systematic integration of GIST into a DBMS. It bridges the gap between the relation and state worlds and supports applications that require both data-parallel and state-parallel computations. We exemplify the use of GIST abstraction through three high-impact machine learning algorithms and show thorough experimental evaluation that the DBMS UDA–GIST can outperform the state-of-the-art by orders of magnitude. We also show that in general with UDA–GIST, we can unify batch and query-time inference efficiently in database.

Acknowledgements This work was partially supported by NSF IIS Award No. 1526753, DARPA under FA8750-12-2-0348-2 (DEFT/CUBISM), and a generous gift from Google.

References

1. Arumugam, S., Dobra, A., Jermaine, C.M., Pansare, N., Perez, L.L.: The datapath system: a data-centric analytic processing engine for large data warehouses. In: Elmagarmid, A.K., Agrawal, D. (eds.) SIGMOD Conference, pp. 519–530. ACM (2010)
2. Bagga, A., Baldwin, B.: Entity-based cross-document coreferencing using the vector space model. In: Boitet, C., Whitelock, P. (eds.) COLING-ACL, pp. 79–85. Morgan Kaufmann Publishers/ACL (1998)
3. Bain, T., Davidson, L., Dewson, R., Hawkins, C.: User defined functions. In: SQL Server 2000 Stored Procedures Handbook, pp. 178–195. Springer, New York (2003)
4. Beedkar, K., Del Corro, L., Gemulla, R.: Fully parallel inference in Markov Logic networks. In: BTW, pp. 205–224. Citeseer (2013)
5. Carlson, A., Betteridge, J., Kisiel, B., Settles, B., Hruschka, E.R. Jr, Mitchell, T.M.: Toward an architecture for never-ending language learning. In: AAAI, vol. 5, p. 3 (2010)
6. Casella, G., George, E.I.: Explaining the Gibbs sampler. Am. Stat. **46**(3), 167–174 (1992)
7. Chafi, H., Sujeeth, A.K., Brown, K.J., Lee, H., Atreya, A.R., Olukotun, K.: A domain-specific approach to heterogeneous parallelism. SIGPLAN Not. **46**(8), 35–46 (2011)
8. Chechetka, A., Guestrin, C.: Focused belief propagation for query-specific inference. In: International Conference on Artificial Intelligence and Statistics, pp. 89–96 (2010)
9. Chen, Y., Wang, D.Z.: Knowledge expansion over probabilistic knowledge bases. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. SIGMOD '14, pp. 649–660. ACM, New York, NY, USA (2014)
10. Chib, S., Greenberg, E.: Understanding the Metropolis–Hastings algorithm. Am. Stat. **49**(4), 327–335 (1995)
11. Cohen, J., Dolan, B., Dunlap, M., Hellerstein, J.M., Welton, C.: Mad skills: new analysis practices for big data. PVLDB **2**(2), 1481–1492 (2009)
12. Cohen, S.: User-defined aggregate functions: bridging theory and practice. In: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, pp. 49–60. ACM (2006)
13. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI, pp. 137–150. USENIX Association (2004)
14. Dobra, A.: Datapath: high-performance database engine, June (2011)
15. Fader, A., Soderland, S., Etzioni, O.: Identifying relations for open information extraction. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, pp. 1535–1545. Association for Computational Linguistics (2011)
16. Felzenszwalb, P.F., Huttenlocher, D.P.: Efficient belief propagation for early vision. CVPR **1**, 261–268 (2004)
17. Friedman, N., Geiger, D., Goldszmidt, M.: Bayesian network classifiers. Mach. Learn. **29**(2–3), 131–163 (1997)
18. Hellerstein, J.M., Ré, C., Schoppmann, F., Wang, D.Z., Fratkin, E., Gorajek, A., Ng, K.S., Welton, C., Feng, X., Li, K., Kumar, A.: The madlib analytics library or mad skills, the sql. CoRR, arXiv:1208.4165 (2012)
19. Hellerstein, J.M., Ré, C., Schoppmann, F., Wang, D.Z., Fratkin, E., Gorajek, A., Ng, K.S., Welton, C., Feng, X., Li, K., Kumar, A.: The madlib analytics library: or MAD skills, the SQL. Proc. VLDB Endow. **5**(12), 1700–1711 (2012)
20. Ihler, A.T., Iii, J., Willsky, A.S.: Loopy belief propagation: convergence and effects of message errors. J. Mach. Learn. Res. 905–936 (2005)
21. Jiang, S., Lowd, D., Dou, D.: Learning to refine an automatically extracted knowledge base using Markov Logic. In: ICDM, pp. 912–917 (2012)
22. Kok, S., Singla, P., Richardson, M., Domingos, P., Sumner, M., Poon, H., Lowd, D.: The Alchemy System for Statistical Relational AI. University of Washington, Seattle (2005)
23. Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. MIT Press, Cambridge (2009)
24. Lao, N., Mitchell, T., Cohen, W.W.: Random walk inference and learning in a large scale knowledge base. In: Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing, pp. 529–539, Edinburgh, Scotland, UK., July. Association for Computational Linguistics (2011)
25. Li, K., Grant, C., Wang, D.Z., Khatri, S., Chitouras, G.: Gptext: Greenplum parallel statistical text analysis framework. In: Proceedings of the Second Workshop on Data Analytics in the Cloud, pp. 31–35. ACM (2013)
26. Li, K., Wang, D.Z., Dobra, A., Dudley, C.: UDA–GIST: An in-database framework to unify data-parallel and state-parallel analytics. In: Proceedings of the VLDB Endowment, vol. 8, no. 5 (2015)
27. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., J.M. Hellerstein. Graphlab: A new framework for parallel machine learning. CoRR, arXiv:1006.4990 (2010)
28. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Distributed GraphLab: a framework for machine learning in the cloud. PVLDB **5**(8), 716–727 (2012)
29. Mahout, A.: Scalable machine-learning and data-mining library. Available at mahout. apache. org
30. Meng, J., Chakradhar, S., A.R. Best-effort parallel execution framework for recognition and mining applications. In: IEEE International Symposium on Parallel Distributed Processing, 2009. IPDPS 2009, pp. 1–12, May (2009)

31. Mitchell, T., Cohen, W.: Data sets and supplementary files (2010). Online; accessed 5 Mar 2015
32. Mitzenmacher, M.: The power of two choices in randomized load balancing. IEEE Trans. Parallel Distrib. Syst. **12**(10), 1094–1104 (2001)
33. Murphy, K.P., Weiss, Y., Jordan, M.I.: Loopy belief propagation for approximate inference: an empirical study. In: Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence, pp. 467–475. Morgan Kaufmann Publishers Inc. (1999)
34. Niepert, M., Domingos, P.M.: Tractable probabilistic knowledge bases: Wikipedia and beyond. In: AAAI Workshop: Statistical Relational Artificial Intelligence (2014)
35. Niu, F., Ré, C., Doan, A., Shavlik, J.: Tuffy: scaling up statistical inference in markov logic networks using an RDBMS. Proc. VLDB Endow. **4**(6), 373–384 (2011)
36. Poon, H., Domingos, P.: Sound and efficient inference with probabilistic and deterministic dependencies. AAAI **6**, 458–463 (2006)
37. Richardson, M., Domingos, P.: Markov logic networks. Mach. Learn. **62**(1–2), 107–136 (2006)
38. Rozanov, Y.A.: Markov Random Fields. Springer, New York (1982)
39. Rusu, F., Dobra, A.: Glade: a scalable framework for efficient analytics. Oper. Syst. Rev. **46**(1), 12–18 (2012)
40. Schoenmackers, S., Etzioni, O., Weld, D.S., Davis, J.: Learning first-order horn clauses from web text. In: Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing, pp. 1088–1098. Association for Computational Linguistics (2010)
41. Sen, P., Deshpande, A., Getoor, L.: Prdb: managing and exploiting rich correlations in probabilistic databases. VLDB J. Int. J. Very Large Data Bases **18**(5), 1065–1090 (2009)
42. Shin, J., Wu, S., Wang, F., De Sa, C., Zhang, C., Ré, C.: Incremental knowledge base construction using deepdive. Proc. VLDB Endow. **8**(11), 1310–1321 (2015)
43. Singh, S., Subramanya, A., Pereira, F., McCallum, A.: Wikilinks: A large-scale cross-document coreference corpus labeled via links to Wikipedia. Technical Report UM-CS-2012-015 (2012)
44. Singh, S., Subramanya, A., Pereira, F.C.N., McCallum, A.: Large-scale cross-document coreference using distributed inference and hierarchical models. In: Lin, D., Matsumoto, Y., Mihalcea, R. (eds.) ACL, pp. 793–803. The Association for Computer Linguistics (2011)
45. Smullyan, R.M.: First-Order Logic, vol. 21968. Springer, Berlin (1968)
46. Sümer, Ö., Acar, U.A., Ihler, A.T., Mettu, R.R.: Adaptive exact inference in graphical models. J. Mach. Learn. Res. **12**, 3147–3186 (2011)
47. Wang, D.Z., Chen, Y., Grant, C., Li, K.: Efficient in-database analytics with graphical models. IEEE Data Eng. Bull. **37**, 41–51 (2014)
48. Wang, H., Zaniolo, C.: User defined aggregates in object-relational systems. In: Proceedings of 16th International Conference on Data Engineering, 2000, pp. 135–144 (2000)
49. Wei, W., Erenrich, J., Selman, B.: Towards efficient sampling: exploiting random walk strategies. AAAI **4**, 670–676 (2004)
50. Wick, M., McCallum, A., Miklau, G.: Scalable probabilistic databases with factor graphs and mcmc. Proc. VLDB Endow. **3**(1–2), 794–804 (2010)
51. Wick, M.L., McCallum, A.: Query-aware MCMC. In: Advances in Neural Information Processing Systems, pp. 2564–2572 (2011)
52. Wikipedia. Hierarchical and recursive queries in SQL (2014). Online; accessed 25 Jan 2015
53. Wikipedia. Barack obama citizenship conspiracy theories (2015). Online; Accessed 25 Jan 2015
54. Xin, R.S., Gonzalez, J.E., Franklin, M.J., Stoica, I.: Graphx: A resilient distributed graph system on spark. In: First International Workshop on Graph Data Management Experiences and Systems, p. 2. ACM (2013)
55. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, pp. 10 (2010)