Towards High-Throughput Gibbs Sampling at Scale: A Study across Storage Managers

Ce Zhang Christopher Ré
University of Wisconsin-Madison, USA
{czhang,chrisre}@cs.wisc.edu

ABSTRACT

Factor graphs and Gibbs sampling are a popular combination for Bayesian statistical methods that are used to solve diverse problems including insurance risk models, pricing models, and information extraction. Given a fixed sampling method and a fixed amount of time, an implementation of a sampler that achieves a higher throughput of samples will achieve a higher quality than a lower-throughput sampler. We study how (and whether) traditional data processing choices about materialization, page layout, and buffer-replacement policy need to be changed to achieve high-throughput Gibbs sampling for factor graphs that are larger than main memory. We find that both new theoretical and new algorithmic techniques are required to understand the tradeoff space for each choice. On both real and synthetic data, we demonstrate that traditional baseline approaches may achieve two orders of magnitude lower throughput than an optimal approach. For a handful of popular tasks across several storage backends, including HBase and traditional unix files, we show that our simple prototype achieves competitive (and sometimes better) throughput compared to specialized state-of-the-art approaches on factor graphs that are larger than main memory.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management; H.3.2 [**Information System**]: Information Storage and Retrieval—*Information Storage*

General Terms

Experimentation, Performance

Keywords

Gibbs sampling; storage manager; scalability

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

1. INTRODUCTION

Factor graphs capture and unify a range of data-driven statistical tasks in information extraction, text analytics, predictive analytics, and pricing and actuarial models that are used across many industries. For example, conditional random fields (CRF) are factor graphs that are used in industry for information extraction [22, 42, 43], e.g., Microsoft's EntityCube [46] and HP's IsWiki [9]. Moody's rating service uses Gibbs sampling and generalized linear models (GLMs) [39], as do leading insurance actuaries [4, 13]. Increasingly, more sophisticated factor graph models are used. For example, Yahoo! and Google use latent Dirichlet allocation (LDA) for topic modeling [8, 23, 38], and Microsoft's EntityCube uses Markov logic [12, 28, 36, 46]. Most inference or parameter estimation problems for factor graphs are intractable to solve exactly, e.g., segmenting a sentence using a skip-chain CRF, inverting the parameters of a GLM to estimate the risk of an actuarial model, or computing the parameters of an LDA model. To perform these tasks, one often resorts to sampling. Arguably the most popular of these approaches, and the one on which we focus, is Gibbs sampling [34].

Not surprisingly, frameworks that combine factor graphs and Gibbs sampling are popular and widely used. For example, the OpenBUGS framework has been used widely since the 1980s [25]. More recent examples include PGibbs [15] and Factorie [26]. For any factor graph, given a fixed time budget, an implementation of Gibbs sampling that produces samples at a higher throughput, achieves higher-quality results. Thus, a key technical challenge is to create an implementation of *high-throughput* Gibbs samplers.

Achieving high throughput for Gibbs sampling is well studied for factor graphs that fit in main memory, but there is a race to apply these approaches to larger amounts of data, e.g., in defense and intelligence [27], enterprise [17], and web applications [38].

A key pain point is that for each specific model, a scalable implementation is hand tuned, e.g., Greenplum's MADlib [17] and Yahoo! [38] implement hand-tuned versions of LDA. High-performance implementations of Gibbs sampling often require a tedious process of trial-and-error optimization. Further complicating an engineer's job is that the variety of storage backends has exploded in the last few years: traditional files, relational databases, and HDFS-based key-value stores, like HBase or Accumulo. The lack of scalable Gibbs samplers forces developers to revisit I/O tradeoffs for each new statistical model and each new data storage combination. We first show that a baseline approach that picks classically

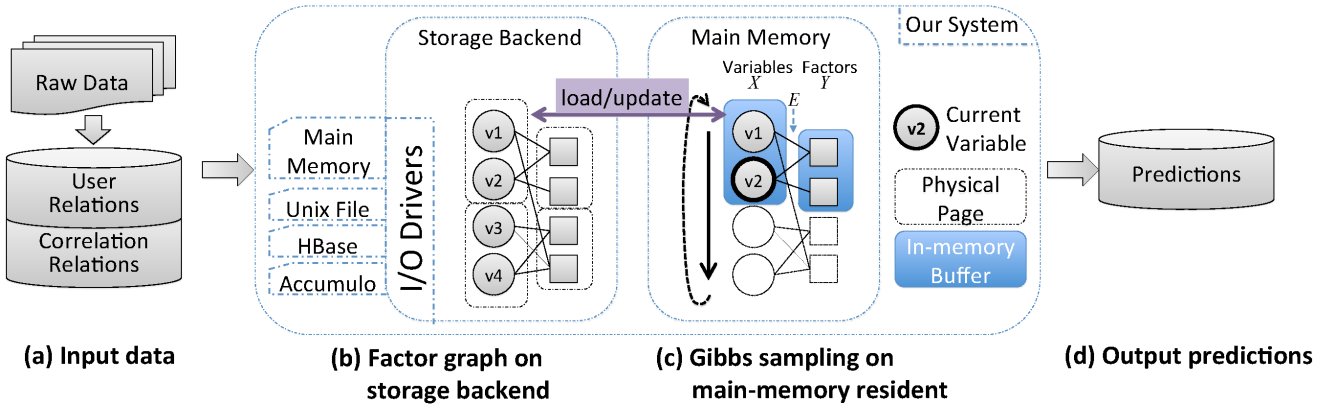

Figure 1: System overview. (a) The input to our system is a factor graph encoded in correlation relations (see Section 2). (b) Factor graphs are stored using several secondary storage backends, e.g., traditional unix file and HBase. (c) Gibbs sampling is performed. Our prototype loads only a portion of the factor graph into main memory, runs inference, and updates the state, which may be stored on secondary storage. (d) As is standard, these samples are summarized for use in the above applications.

optimal points in this tradeoff space may lose almost two orders of magnitude in throughput (see Figures 11 and 14).

We select three classical techniques that are used to improve query performance in database storage managers: materialization, page-oriented layout, and buffer-replacement policy. These techniques are simple to implement (so they may be implemented in practice), and they have stood the test of four decades of use.[1] Our study examines these techniques for a handful of popular storage backends: main-memory based, traditional unix files, and key-value stores.

Applying these classical techniques is not as straightforward as one may think. The tradeoff space for each technique has changed due to at least one of two new characteristics of sampling algorithms (versus join data processing): (1) data are mutable during processing (in contrast to a typical join algorithm), and (2) data are repeatedly accessed in an order that changes randomly across executions, which means that we cannot precompute an ideal order. However, within any given sampling session, the data is accessed in a fixed order. We consider the tradeoff space analytically, and use our analysis of the tradeoff space to design a simple proof-of-concept storage manager. We use this storage manager to validate our claimed tradeoff space in scaling Gibbs sampling.

As we show through experiments, our prototype system is competitive with even state-of-the-art, hand-tuned approaches for factor graphs that are larger than main memory. Thus, it is possible to achieve competitive performance using this handful of classical techniques.

Overview of Technical Contributions

In Section 2, we describe a representation of factor graphs that follows the work of Sen et al. [37] and Wick et al. [44].[2] We give a brief primer on factor graphs to provide context for our contributions, which we discuss at the end of this section.

A Primer of Gibbs Sampling on Factor Graphs. The input to Gibbs sampling is a factor graph, which is a labeled bipartite graph $G = (X, Y, E)$ where X and Y are sets of nodes and $E \subseteq X \times Y$ is a set of edges (illustrated in Figure 1). In a factor graph, X contains one node for each random variable, which is labeled with the current assignment to that variable, and Y contains a node for each factor. The factors define the correlation structure between the variables: two variables are connected to the same factor if they are correlated. A *factor function* is associated with each factor and numerically describes how sets of variables that are related by a factor are correlated. E describes which variables are associated with which factors.

The output of sampling is one or more samples for each variable in the factor graph. To obtain a sample for a single variable x, we perform three steps: (1) retrieve all neighboring factors and the assignments for all variables in those factors, (2) evaluate these neighboring factors with the retrieved assignments, and (3) aggregate the evaluation results to compute a conditional distribution from which we select a new value for x. This operation, which we call the *core operation*, is repeated for every variable in the factor graph (in a randomly selected order). We loop over the variables many times to obtain a high-quality sample.

Technical Contributions. Since the bulk (often over 99%) of the execution time is spent in the core operation, we focus on optimizing this operation. To this end, we identify three tradeoffs for the core operation of Gibbs sampling: (1) materialization, (2) page-oriented layout, and (3) buffer-replacement policy.

(1) Materialization tradeoffs. During Gibbs sampling, we access the bipartite graph structure many times. An obvious optimization is to materialize a portion of the graph to avoid repeated accesses. A twist is that the value of a random variable is updated during the core operation. Thus, materialization schemes that introduce redundancy may re-

[1] Although sampling approaches have been integrated with data processing systems, notably the MCDB project [20], the above techniques have not been applied to the area of high-throughput samplers.

[2] A minor difference is that we do not require, as Sen et al. [37] or PGibbs does, that one explicitly list the complete truth table for every factor (see Section 5).

Method	Comment
`void init(𝔻[] vars)`	initialize this factor with a list of variable IDs
`float evaluate()`	evaluate the value of this factor

Figure 2: Interface for a factor function. Variable assignments are globally accessible.

quire many (random) writes. This introduces a new cost in materialization schemes, and so we re-explore lazy and eager materialization schemes, along with two co-clustering schemes [33, ch. 16]. The two co-clustering schemes together analytically and empirically dominate both eager and lazy schemes – but neither dominates the other. Thus, we design a simple optimizer based on statistics from the data to choose between these two approaches.

(2) Page-oriented layout. Once we have decided how to materialize or co-cluster, we are still free to assign variables and factors to pages to minimize the number of page fetches from the storage manager during the core operation. In cases in which the sampling is I/O bound, minimizing the number of page fetches helps reduce this bottleneck. Since Gibbs sampling proceeds through the variables in a random order, we want a layout that has good average-case performance. Toward this goal, we describe a simple heuristic that outperforms a naïve random layout by up to an order of magnitude. We then show that extending this heuristic is difficult: assuming P $\neq$ NP, no polynomial time algorithm has a constant-factor approximation for the layout that minimizes page fetches during execution.

(3) Buffer-Replacement Policy. A key issue, when dealing with data that is larger than available main memory, is deciding which data to retain and which to discard from main memory, i.e., the buffer-replacement policy. Although the data is visited in random order, one will typically perform many passes over the data in the same random order. Thus, after the first pass over the data, we have a great deal of information about which pages will be selected during execution. It turns out that this enables us to use a classical result from Bélády in 1966 [7] to define a theoretically optimal eviction strategy: *evict the item that will be used latest in the future*. We implement this approach and show that it performs optimally, but that its improvement over more traditional strategies based on MRU/LRU is only 5–10%. It also requires some non-trivial implementation, and so it may be too complex for the performance gain. A technical serendipity is that we use Bélády's result to characterize an optimal buffer-replacement strategy, which is needed to establish our hardness result described in page-oriented layout.

We present background information on factor graphs and Gibbs sampling in Section 2, our technical contributions in Section 3, and an empirical validation in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2. BACKGROUND

We describe factor graphs, the mechanics of Gibbs sampling, and how Gibbs sampling supports inference and query answering.

2.1 Factor Graphs

Factor graphs have been used to model complex correlations among random variables for decades [41], and have recently been adopted as one underlying representation for probabilistic databases [37, 42, 44]. A key principle of factor graphs is that random variables should be separated from the correlation structure. Following this principle, we define a probabilistic database to be $\mathcal{D} = (\mathcal{R}, \mathcal{F})$, where $\mathcal{R}$ is called the *user schema* and $\mathcal{F}$ is called the *correlation schema*. Relations in the user schema (resp. correlation schema) are called *user relations* (resp. *correlation relations*). A user's application interacts with the user schema, while the correlation schema captures correlations among tuples in the user schema. Figure 3 gives a schematic view of the concepts in this section.

User Schema. Each tuple in a user relation $R_i \in \mathcal{R}$ has a unique tuple ID, which takes values from a *variable ID domain* $\mathbb{D}$, and is associated with a random variable[3] taking values from a *variable value domain* $\mathbb{V}$. We assume that $\mathbb{V}$ is discrete (e.g., Boolean).[4] Each distinct *variable assignment* $\sigma : \mathbb{D} \mapsto \mathbb{V}$ defines a *possible world* I_σ that is equivalent to a standard database instantiation of the user schema.

Correlation Schema. The correlation schema defines a probability distribution over the possible worlds as follows. Intuitively, each correlation relation $F_j \in \mathcal{F}$ represents one type of correlation over the random variables in $\mathbb{D}$ by specifying *which* variables are correlated and *how* they are correlated. To specify *which*, the schema of F_j has the form $F_j(\underline{fid}, \bar{v})$ where fid is a unique factor ID taking values from a *factor ID domain* $\mathbb{F}$, and $\bar{v} \in \mathbb{D}^{a_j}$ where a_j is the number of random variables that are correlated by each factor.[5] In Figure 3, variables v_1 and v_4 are correlated. To specify *how*, F_j is associated with a function $f_j : \mathbb{V}^{a_j} \mapsto \mathbb{R}_+$. Given a possible world I_σ, for any $t = (fid, v_1, \ldots, v_{a_j}) \in F_j$, define $g_j(t, I_\sigma) = f_j(\sigma(v_1), \ldots, \sigma(v_{a_j}))$. Define $\text{vars}(fid) = \{v_1, \ldots, v_{a_j}\}$. To explain independence in the graph, we need the notion of the *Markov blanket* [30, 41] of a variable v, denoted $\mathbf{mb}(v)$, and defined to be

$$\mathbf{mb}(v_i) = \{v \mid v \neq v_i,\ \exists fid \in \mathbb{F} \text{ s.t. } \{v, v_i\} \subseteq \text{vars}(fid)\}.$$

In Figure 3, $\mathbf{mb}(v_1) = \{v_2, v_4\}$. In general, a variable v is conditionally independent of a variable $v' \notin \mathbf{mb}(v)$ given $\mathbf{mb}(v)$. Here, v_1 is independent of v_3 given $\{v_2, v_4\}$.

To specify a factor function in our prototype, the user implements the C++ interface in Figure 2.[6]

Probability Distribution. Denote by $\mathcal{I}$ the set of all possible worlds. Define the *partition function* $Z : \mathcal{I} \mapsto \mathbb{R}_+$ over

[3]Either modeling the existence of a tuple (for Boolean variables) or in the form of a special attribute of R_i. The value of a random variable may be fixed as input.

[4]We describe factor graphs over continuous variables in the full version of this paper.

[5]Our representation supports factors with varying arity as well, but we focus on fixed-arity factors for simplicity.

[6]Our infrastructure also accepts higher level languages, e.g., OpenBUGS or Markov Logic, but this compilation is orthogonal to our contributions.

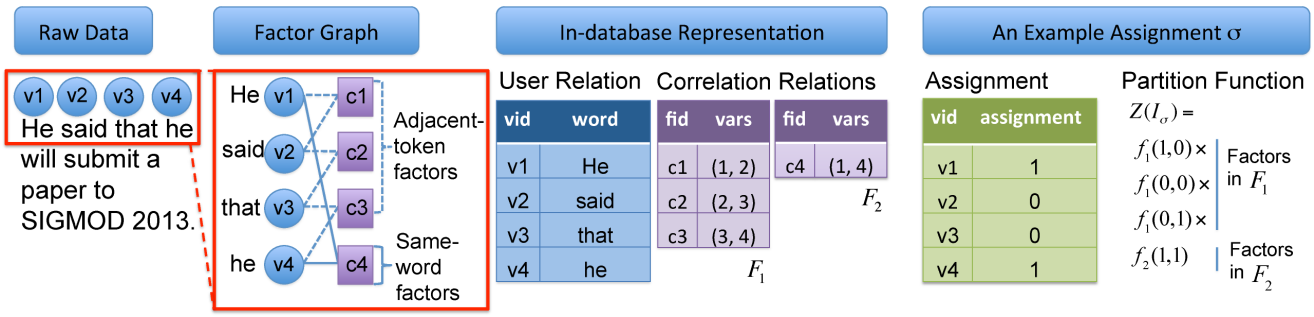

Figure 3: An example factor graph that represents a simplified skip-chain CRF; more sophisticated versions of this model are used in named-entity recognition and parsing. There is one user relation containing all tokens, and there are two correlation relations for adjacent-token correlation (F1) and same-word correlation (F2) respectively. The factor function associated to F1 (resp. F2), called f_1 (resp. f_2), is stored in a manner similar to a user-defined function.

any possible world $I \in \mathcal{I}$ as

$$Z(I) = \prod_{F_j \in \mathcal{F}} \prod_{t \in F_j} g_j(t, I) \tag{1}$$

The probability of a possible world $I \in \mathcal{I}$ is

$$\Pr[I] = Z(I) \left(\sum_{J \in \mathcal{I}} Z(J) \right)^{-1}. \tag{2}$$

We assume that $\sum_{J \in \mathcal{I}} Z(J) > 0$. It is well known that this representation can encode all discrete probability distributions over possible worlds [6].

2.2 Inference and Query Answering

In a factor graph, which defines a probability distribution, (marginal) *inference* refers to the process of computing the probability that a random variable will take a particular value. Marginal inference on factor graphs is a powerful framework. Following Sen et al. [37], we can use it to support sophisticated query answering via a two-stage process: (1) we first add a new user relation that describes the output of the query, and (2) we add in correlation relations as described by Sen et al.'s algorithm. This process creates a new factor graph, on which we perform marginal inference.

Gibbs Sampling. As exact inference for factor graphs is intractable, a commonly used inference approach is to sample, and then use the samples to compute the marginal probabilities of random variables (e.g., by averaging the outcomes). The main idea of Gibbs sampling is as follows. We start with a random possible world I_0. For each variable $v \in \mathbb{D}$, we sample a new value of v according to the conditional probability $\Pr[v|\mathbf{mb}(v)]$. A routine calculation shows that computing this probability requires that we retrieve the assignment to the variables in $\mathbf{mb}(v)$ and the factor functions associated with the factor nodes that neighbor v. Then we update v in I_0 to the new value. After scanning all variables, we obtain a first sample I_1. We repeat this process to generate I_{k+1} from I_k.

Example We continue our example (see Figure 3) of extracting information by labeling each token in a sentence using a skip-chain conditional random field [40], a type of

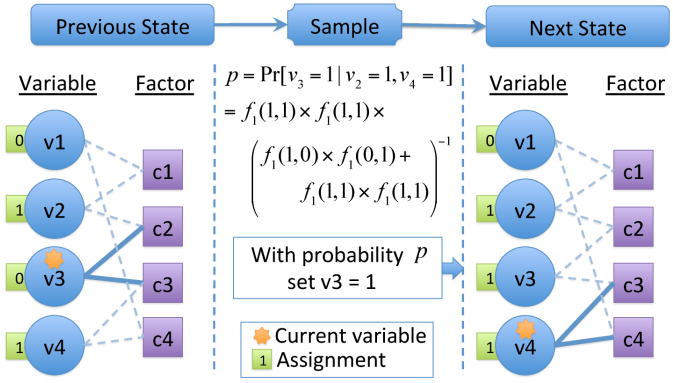

Figure 4: A step of Gibbs sampling. To sample a variable v_3, we read the current values of variables in $\mathbf{mb}(v_3) = \{v_2, v_4\}$ and calculate the conditional probability $\Pr[v_3|v_2, v_4]$. Here, the sample result is $v_3 = 1$. We update the value of v_3 and then proceed to sample v_4.

factor graphs used in named-entity recognition [14]. Figure 4 illustrates one step in Gibbs sampling to update a single variable, v_3: we compute the probability distribution over v_3 by retrieving factors c_2 and c_3, which determines $\mathbf{mb}(v_3)$. Then, we find the values of v_2 and v_4. Using Equation 2, one can check that the probability that $v_3 = 1$ is given by the equation in Figure 4 (conditioned on $\mathbf{mb}(v_3)$, the remaining factors cancel out).

There are several popular optimization techniques for Gibbs sampling that we implement in our prototype. One class of techniques improves the sample quality, e.g., *burn-in*, in which we discard the first few samples, and *thinning*, in which we retain every kth sample for some integer k [3]. A second class of techniques improves throughput using parallelism. The key observation is that variables that do not share factors may be sampled independently and so in parallel [45]. These techniques are essentially orthogonal to our contributions, and we do not discuss them further.

3. STORAGE-MANAGER TRADEOFFS TO SCALE GIBBS SAMPLING

We show that Gibbs sampling essentially performs joins and updates on a view over several base relations that represent the factor graph. Thus, we can apply classical data management techniques (such as view materialization, page-oriented layout, and buffer-replacement policy) to achieve scalability and high I/O efficiency.

3.1 Gibbs Sampling as a View

Given a database with a user schema and correlation schema as defined in the previous section, we define two simple relations that represent the edge relation of the factor graph and the sampled possible world:

$$E(v, f) = \{(v, f) | f \in \mathbb{F},\ v \in \text{vars}(f)\} \text{ and } A(\underline{v}, a) \subseteq \mathbb{D} \times \mathbb{V}$$

In Figure 3, E encodes the bipartite graph and each tuple in A corresponds to a possible assignment of a label of a variable. Note that each variable is assigned a single value (so v is a key of A) and A will be modified during Gibbs sampling.

The following view describes the input factor graph for Gibbs sampling:

$$Q(v, f, v', a') \leftarrow E(v, f), E(v', f), A(v', a'), v \neq v'.$$

When running Gibbs sampling, we group Q by the first field v. Notice that the corresponding group of tuples in Q contains all variables in the Markov blanket $\mathbf{mb}(v)$ along with all the factor IDs incident to v. This is all the information we need to generate a sample. For each group, corresponding to a variable v, we sample a new value (denoted by a) for v according to the conditional probability $\Pr[v|\mathbf{mb}(v)]$. The twist is that after we obtain a, we must update A before we proceed to the next variable to be sampled. In our implementation, we proceed through the groups in a random order.[7]

After one pass through Q, we obtain a new possible world (represented by A). We repeat this process to obtain multiple samples. By counting the number of occurrences of an event, e.g., a variable taking a particular value, we can use these samples to perform marginal inference.

We describe how we optimize the data access to Q using classical data management techniques such as view materialization and page-oriented layout. To study these tradeoffs, we implement a simple storage system. We store E and A using slotted pages [33, ch. 16] and a single-pool buffer manager.[8]

3.2 Materializing the Factor Graph

The view Q involves a three-way join. All join operators perform an index nested-loop join (see Figure 6) with a small twist: the join keys are also *record IDs*, hence we do not need to perform an actual index look-up to find the appropriate page. We examine materialization strategies to improve the performance of these joins.[9]

Materialization can reduce the number of random reads. However, a key twist is that the base relation A is updated after we examine each group. Thus, if we replicate A we could incur many random writes. We study this tradeoff by comparing four different materialization strategies in terms of random read/write costs and space requirements (see Figure 5).

- **Lazy:** we only materialize base tables. This strategy incurs the most random reads for on-the-fly joins, but the fewest random writes.
- **V-CoC:** we co-cluster on the variable side

  $$QV(v, f, v') \leftarrow E(v, f), E(v', f), v \neq v'$$

  Compared to the lazy strategy, this strategy eliminates random reads on E and retains a single copy of A.
- **F-CoC:** we co-cluster on the factor side

  $$QF(f, v', a) \leftarrow E(v', f), A(v', a)$$

  Compared to the lazy strategy, this strategy eliminates random reads on A at the cost of more random writes to update A within QF.
- **Eager:** we eagerly materialize Q. This strategy eliminates all random reads at the cost of high random writes during the core operation.

Figure 6 illustrates the two co-clustering strategies.

	Reads from A	Reads from E	Writes A	Storage
Lazy	$\lvert\mathbf{mb}(v)\rvert$	d_v	1	$O(\lvert E\rvert + \lvert A\rvert)$
V-CoC	$\lvert\mathbf{mb}(v)\rvert$	0	1	$O(\sum_v \lvert\mathbf{mb}(v)\rvert + \lvert A\rvert)$
F-CoC	0	d_v	d_v	$O(\lvert E\rvert)$
Eager	0	0	$\lvert\mathbf{mb}(v)\rvert$	$O(\sum_v \lvert\mathbf{mb}(v)\rvert)$

Figure 5: I/O costs for sampling one variable under different materialization strategies. For a variable v, d_v is the number of factors in which v participates. Reads (resp. writes) are random reads (resp. random writes). The cost for each strategy is the total number of reads and writes.

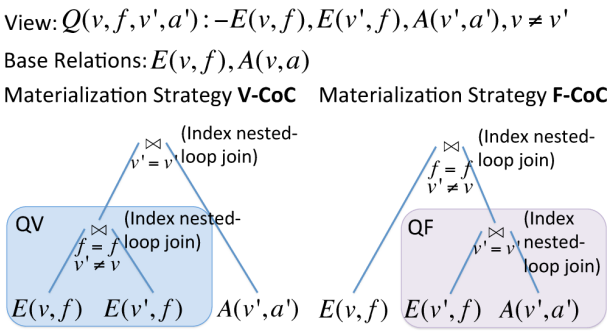

Figure 6: Strategies V-CoC and F-CoC.

[7] Our strategy is independent of the order. The order, however, empirically does change the convergence rate, and it is future work to understand the tradeoff of order and convergence.

[8] We use variable and factor IDs as record IDs, and so can look up the column v or f in both E and A using one random access (similar to RID lists [33, ch. 16]).

[9] Since we run many iterations of Gibbs sampling, the construction cost of materialized views is often smaller than the cost of running Gibbs sampling in all strategies. (In all of our experiments, the construction cost is less than 25% of the total run time.)

From Figure 5, we can see that V-CoC dominates Lazy and F-CoC dominates Eager in terms of random accesses (Assuming each factor contains at least two variables, $|\mathbf{mb}(v)| \geq 2d_v$). Our experiments confirm these analytic models (see Section 4.3). Therefore, our system selects between V-CoC and F-CoC. To determine which approach to use, our system selects the materialization strategy with the smallest random I/O cost according to our cost model.

3.3 Page-oriented Layout

With either V-CoC or F-CoC, Q is an index nested-loop join between one relation clustered on variables and another relation clustered on factors. How the data are organized into pages impacts I/O efficiency in any environment.

We formalize the problem of page-oriented layout for Gibbs sampling as follows, GLAYOUT. The input to the problem is a tuple (V, F, E, μ, S, P) where (V, F, E) define a factor graph; that is, $V = \{v_1, \ldots, v_N\}$ is a set of variables, $F = \{f_1, \ldots, f_M\}$ is the set of factors, and $E = \{(v, f) \mid v \in \text{vars}(f)\}$ defines the edge relation. The order μ is a total order on the variables, V, which captures the fact that at each iteration of Gibbs sampling, one scans the variables in V in a random order, and S the page size (i.e., the maximum number of variables or factors a page can store), and P the set of pages.

Any solution for GLAYOUT is a tuple (α, β, π) where $\alpha : V \mapsto P$ (resp. $\beta : F \mapsto P$) map variables (resp. factors) to pages. The order $\pi = (e_1, \ldots, e_L)$ defines the order of edges that the algorithm visits. That is, given a fixed variable visit order $\mu = (v_1, \ldots, v_N)$, we only consider tuple orders π that respect μ; we denote this set of orders as $\Pi(\mu)$.[10]

As we will see, it is possible in Gibbs sampling to construct an *optimal* page eviction strategy. For the moment, to simplify our presentation, we assume that there are only two buffer pages: one for variables and one for factors. We return to this point later. Then any ordering $\pi = (e_1, \ldots, e_L)$ would incur an I/O cost (for fetching factors from disk):

$$\text{cost}(\pi, \beta) = 1 + \sum_{i=2}^{L} \mathbb{I}\left[\beta(e_i.f) \neq \beta(e_{i-1}.f)\right]$$

The goal is to find an ordering $\pi \in \Pi(\mu)$ and mapping β that minimizes the above I/O cost.

Algorithm. We use the following simple heuristic to construct (α, β, π) ordering F: we sort the factors in F in dictionary order by μ, that is, by the position of the earliest variable that occurs in a given factor. We then greedily pack F-tuples onto disk pages in this order. We show empirically that this layout has one order of magnitude improvement over randomly ordering and paging tuples.

Analysis. Extending this heuristic for better performance is theoretically challenging:

PROPOSITION 3.1. *Assuming* P≠NP*, an algorithm does not exist that runs in polynomial time to find the optimal solution to* GLAYOUT*. More strongly, assuming* P≠NP*, no polynomial time algorithm can even guarantee a constant approximation factor for* cost*.*

We prove this proposition by encoding the multi-cut partition problem [2]. Under slightly more strict complexity assumptions (about randomized complexity classes), it also follows that getting a good layout *on average* is computationally difficult.

Multi-page Buffer. We return to the issue of more than two buffer pages. Since variables are accessed sequentially, MRU is optimal for the variable pages. One could worry that the additional number of buffer pages could be used intelligently to lower the cost. We show that we are able to reduce the multiple-page buffer case to the two-page case (and so an approximation result similar to Proposition 3.1); intuitively, given the instance generated for the two-buffer case, we include extra factors whose role in the reduction is to ensure that any optimal buffering strategy essentially incurs I/O only on the sequence of buffer pages that it would in the two-buffer case. Our proof examines the set of pages in Bélády's [7] optimal strategy, which we describe in the next section.

3.4 Buffer-Replacement Policy

When a data set does not fit in main memory, we must choose which page to evict from main memory. A key twist here is that we will be going over the same data set many times. After the first pass, we know the full reference sequence for the remainder of the execution. This enables us to use a classical result from Bélády in the 1960s [7] to define a theoretically optimal eviction strategy: *evict the item that will be used latest in the future.*

Recall that we scan variables in sequential order, so MRU is optimal for those pages. However, the factor pages require a bit more care. On the first pass of the data, we store the entire reference sequence of factors in a file, i.e., sequence $f_{i_1}, f_{i_2}, \ldots$, lists each factor. For each reference, we then do a pass over this log to compute when each factor is used next in the sequence. At the end of the first scan, we have a log that consists of pairs of factor IDs and when each factor appears next. The resulting file may be several GB in size. However, during the subsequent passes, we only need the head of this log in main memory. We use this information to maintain, for each frame in the buffer manager, when that frame will be used next.

In our experiments, we evaluate under what situations this optimal strategy is worth the extra overhead. We find that the optimal strategy is close to the standard LRU in terms of run time–even if we allow the potentially unfair advantage of holding the entire reference sequence in main memory. In both cases, the number of page faults is roughly 5%–10% smaller using the optimal approach. However, we found that simple schemes based on MRU/LRU are essentially optimal in two tasks (called **LR** and **CRF** in the next section), and in the third task (called **LDA** in the next section) the variables share factors that are far apart, and so even the optimal strategy cannot mitigate all random I/O.

4. EXPERIMENTS

We validate that (1) our system is able to scale to factor graphs that are larger than main memory, and (2) for inference on factor graphs that are larger than main memory, we achieve throughput that is competitive with specialized approaches. We then validate the details of our technical claims about materialization, page-oriented layout, and

[10] $\Pi(\mu) = \{\pi = (e_1, \ldots, e_L) \mid \forall i, j. 1 \leq i \leq j \leq L, e_i.v \preceq_\mu e_j.v\}$, where $x \preceq_\mu y$ indicates that x precedes y in μ.

Systems	LR	CRF	LDA	Storage	Parallelism
Elementary	✓	✓	✓	Multiple	Multi-thread
Factorie	✓	✓	✓	RAM	Single-thread
PGibbs	✓	✓	✓	RAM	Multi-thread
OpenBUGS	✓		✓	RAM	Single-thread
MADlib			✓	Database	Multi-thread

Figure 7: Key features of each system that implements Gibbs sampling. OpenBUGS and MADlib do not support skip-chain CRFs. MADlib supports LR but not a full Bayesian (sampling) treatment; its support for LDA is via specialized UDFs.

buffer-replacement policy that we described in the previous section.

4.1 Experimental Setup

We run experiments on three popular statistical models: 1) logistic regression (**LR**), 2) skip-chain conditional random field [40] (**CRF**), and 3) latent Dirichlet allocation (**LDA**) [11]. We use **LR** and **CRF** for text chunking, and **LDA** for topic modeling. We select **LR** because it can be solved exactly and so can be used as a simple benchmark. We select **CRF** as it is widely used in text-related applications and inference is often performed with Gibbs sampling, and **LDA** as it is intuitively the most challenging model that is supported by all systems in our experiments.

Data Sets. To compare the quality and efficiency with other systems, we use public data sets that we call *Bench*. We run **LR** and **CRF** on CoNLL[12] and **LDA** on AP.[13] For scalability experiments and tradeoff efficiency experiments, we scale up *Bench* from 1x to 100,000x by adding more documents which are randomly selected from a one-day web crawl (5M documents, 400GB). We call the 1,000x data set *Perf* and the 100,000x data set *Scale*. Figure 8 shows the number of variables, factors, and size for each data set.

Metrics. We use two metrics to measure the quality: (1) the F1 score of the final result for data sets with ground truth (when ground truth is available) and (2) the *squared loss* [44] between the marginal probabilities that are the output of each sampling system and a gold standard on each data set (when there is no ground truth). For **LR**, we compute the gold standard using exact inference. For **CRF** and **LDA**, it is intractable to compute the exact distribution; and the standard approach is to use the results from Gibbs sampling after running many iterations beyond convergence [44]. In particular, we get an approximate gold standard for both **CRF** and **LDA** on *Bench* by running Gibbs sampling for 10M iterations. For efficiency, we measure the *throughput* as the number of samples produced by a system per second.

[11] As shown in the full version of this paper, following MADLib [17], we incrementally maintain the evaluation result of LDA factors with regard to changes of variable assignment. We implement this by attaching a fixed-size memory area to each factor. All tradeoffs, i.e., materialization, page-oriented layout, and buffer-replacement policy, still apply to this case.

[12] `http://www.cnts.ua.ac.be/conll2000/chunking/`

[13] `http://www.cs.princeton.edu/~blei/lda-c/`

Setting Name	Page Size	Buffer Size
Sp/Sb	4KB	40KB
Lp/Sb	4MB	40MB
Sp/Lb	4KB	4GB
Lp/Lb	4MB	4GB
Lp/MAXb	4MB	40GB

Figure 9: Different configuration settings of Elementary. Sp (resp. Lp) means *small (resp. large) page size*; Sb (resp. Lb) means *small (resp. large) buffer-size*.

Competitor Systems. We compare our system with four state-of-the-art Gibbs sampling systems: (1) PGibbs on GraphLab [15], (2) Factorie [26], and (3) OpenBUGS [25], which are main memory systems, and MADlib [17], a scalable in-database implementation of LDA. PGibbs and MADlib[14] are implemented using C++, OpenBUGS is implemented using Pascal and C, and Factorie is implemented using Java. We use Greenplum 4.2.1.0 for MADlib. The key features of each system are shown in Figure 7. For each model (i.e., **LR**, **CRF**, and **LDA**), all systems take the same factor graph as their input. We modify each system to output squared loss after each iteration, and we do not count this time as part of the execution time.

Details of our Prototype. We call our prototype system Elementary, and compare three variants of our system by plugging in different storage backends: **EleMM**, **EleFILE**, and **EleHBASE**. We use main memory, traditional unix files, and HBase, respectively, for these three variants. **EleMM** is used to compare with other main memory systems. **EleFILE** and **EleHBASE** use different secondary storage and are used to validate scalability and I/O tradeoffs.

Our system is implemented using C++ and bypasses the OS disk-page cache to support its own buffer manager. We compile all C++ code using GCC 4.7.2 and all Java code using Java 1.7u4. All experiments are run on a RHEL6.1 workstation with two 2.67GHz Intel Xeon CPUs (12 cores, 24 hyper-threaded), 128GB of RAM, and 12×2TB RAID0 drives. All data and temporary files are stored on a RAID, and all software is installed on a separate disk. We use the latest HBase-0.94.1 on a single machine[15] and follow best practices to tune HBase.[16]

We use various page-size/buffer-size combinations, as shown in Figure 9. These configuration settings correspond to small/large page sizes and small/large buffer sizes. We explore more settings in the full version, but the high-level conclusions are similar to these four settings so we omit those results. The **Lp/MAXb** setting is used for end-to-end experiments.

4.2 End-to-end Efficiency and Scalability

We validate that our system achieves state-of-the-art quality by demonstrating that it converges to the gold-standard probability distribution on each task. We then compare our

[14] `http://doc.madlib.net/v0.3/group__grp__plda.html`

[15] Our system can use HBase on multiple machines to get even higher scalability. We only discuss the one-machine case to be fair to other systems.

[16] http://hbase.apache.org/book/performance.html

Models	Bench (1x)			Perf (1,000x)			Scale (100,000x)		
	$\|V\|$	$\|F\|$	Size	$\|V\|$	$\|F\|$	Size	$\|V\|$	$\|F\|$	Size
LR	47K	47K	2MB	47M	47M	2GB	5B	5B	0.19TB
CRF	47K	94K	3MB	47M	94M	3GB	5B	9B	0.3TB
LDA	0.4M	12K	10M	0.4B	10M	9GB	39B	0.2B	0.9TB

Figure 8: Data sets sizes. We omit the 10x, 100x, and 10,000x cases from this table.

	# thread = 1			# threads = 20		
	LR	**CRF**	**LDA**	**LR**	**CRF**	**LDA**
EleMM	24	18	12	12x	9x	6x
EleFILE	34	24	33	9x	6x	4x
EleHBASE	101	70	92	9x	7x	4x
PGibbs	38	42	CRASH	9x	5x	CRASH
MADLib	N/A	N/A	17	N/A	N/A	8x
Factorie	95	120	6	N/A	N/A	N/A
OpenBUGS	150	N/A	CRASH	N/A	N/A	N/A

Figure 10: The time in seconds (speedup for 20-threads case) needed to achieve squared loss below 0.01 for *Bench* data set. N/A means that the task is not supported.

system's efficiency and scalability with that of our competitors.

Quality. We validate that our system converges to the gold-standard probability distribution on several tasks. We run all systems on *Bench* using the configuration **Lp/MAXb**. We take a snapshot after each system produces a sample; we output the squared loss and F1 score of that snapshot. Figure 10 shows the time that each system needs to converge to a solution with a squared loss less than 0.01. We chose 0.01 since a smaller loss does not improve the F1 score, i.e., when the loss is less than 0.01, **LR** converges to F1=0.79 and **CRF** to F1=0.81, which are the gold-standard values for these tasks. This validates that all systems achieve the same quality, and thus a more interesting question is to understand the rate at which they achieve the quality.

Efficiency. To understand the rate at which a system achieves the gold-standard quality, we run PGibbs, MADlib, Factorie, and OpenBUGS and measure when they are within 0.01 of the optimal loss. If a system can take advantage of multi-threading, we also run it with 20 threads and report the speedup factor compared with one thread. We chose our materialization strategy to be consistent with what is hard coded in state-of-the-art competitors, i.e., V-CoC for **LR** and **CRF**, and F-CoC for **LDA**. Figure 10 shows the results.

When there is 1-thread and all the data fit in main memory, we observe that all systems converge to gold-standard probability distribution within roughly comparable times. This is likely due to the fact that main memory Gibbs sampling uses a set of well-known techniques, which are widely implemented. There are, however, some minor differences in runtime. On **LR** and **CRF**, EleMM has comparable efficiency to PGibbs, but is slightly faster. After reviewing PGibbs' code, we believe the efficiency difference (<2x) is caused by the performance overhead of PGibbs explicitly representing factor functions as a table. For **LR** and **CRF**, EleMM is 4–10x faster than Factorie, and we believe that this is because Factorie is written in Java (not C++). OpenBUGS is more than one order of magnitude slower than EleMM. This difference in OpenBUGS is due to the inefficiency in how it records the samples of each variable. PGibbs requires explicit enumeration of all outcomes for each factor, and so the input size to PGibbs is exponential in the size of the largest factor. As **LDA** involves factors with hundreds or more variables, we could not run **LDA** on PGibbs. OpenBUGS crashed on **LDA** while compiling the model and data with the error message "illegal memory read." Elementary with secondary storage backends has similar performance trends, but is overall 2–5x slower than EleMM, due to the overhead of I/O.

For 20-thread cases, we observe speedups for all multi-threading systems. EleFILE and EleHBASE gets smaller speedups compared with EleMM because of the overhead of I/O requests. PGibbs also gets slightly smaller speedups than EleMM, and we believe that this marginal difference is caused by the overhead of MPI, which is used by PGibbs.

Scalability. We validate that our system can scale to larger data sets by using secondary storage backends. We run all systems on data sets scaling from 1x (*Bench*) to 100,000x (*Scale*) using one thread and **Lp/MAXb**. We let each system run for 24 hours and stop it afterwards. Figure 11 shows the results for **LR** and **LDA**.[17] We also run a baseline system, **Baseline**, that uses the file system as storage and disables all of our optimizations (lazy materialization, and no buffer management).

Not surprisingly, only systems that use secondary storage backends, i.e., EleFILE, EleHBASE, and MADlib (only for LDA), scale to the largest data sets; in contrast, main memory systems crash or thrash at scaling factors of 10x or more. As the core operation is scan based, it also not surprising that each system scales up almost linearly (if it is able to run).

However, the performance of the various approaches can be quite different. The Baseline system is three orders of magnitude slower than EleMM (it does not finish the first iteration within 24 hours on the 100x scale factor). This shows that buffer management is crucial to implementing an

[17] **CRF** is similar to **LR**, we include it in the full version.

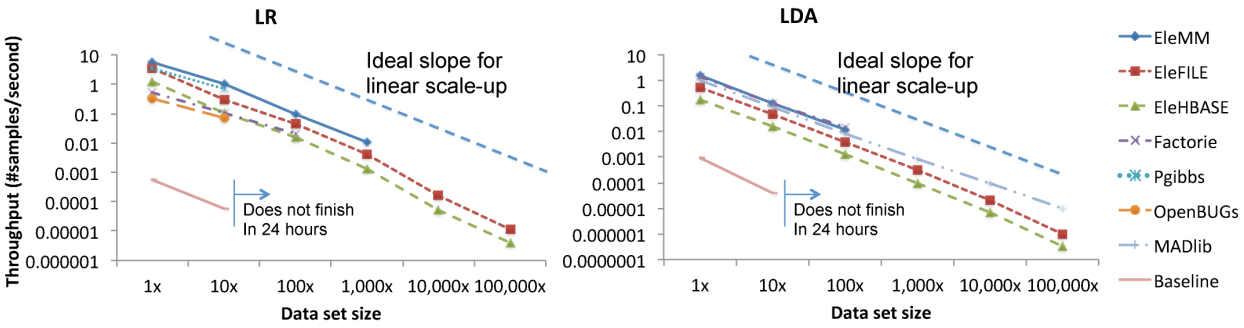

Figure 11: Scalability of Different Systems.

Model	A	E	QV	QF	Q
LR	0.18	1.4	1.4	1.4	1.4
CRF	0.18	2.0	3.2	2.0	3.2
LDA	1.66	9.0	100T+	9.0	100T+

Figure 12: Size of intermediate state in *Perf* dataset (see Sec. 3). All sizes in GB, except where noted.

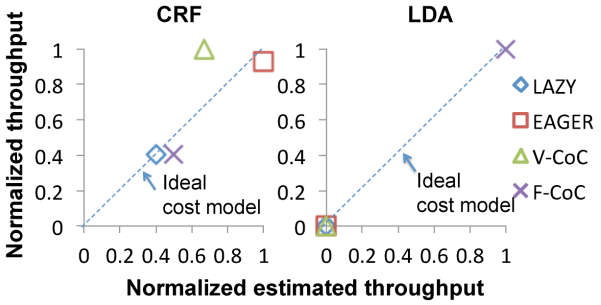

Figure 13: Accuracy of the cost model of different materialization strategies.

I/O-efficient Gibbs sampler. Due to higher I/O overheads, EleHBASE is 3x slower than EleFILE.

MADlib uses PostgreSQL as the storage backend, and is 2x faster than EleFILE. However, MADlib uses an approximate sampler for LDA, while our system implements an exact sampler. If we allow EleFILE to run this approximate sampler, EleFILE is between 10-20% faster.

4.3 I/O tradeoffs

We validate that both the materialization tradeoff and the page-oriented layout tradeoff in Section 3 affect the throughput of Gibbs sampling on factor graphs that are larger than main memory, while the buffer-replacement tradeoff only has a marginal impact. We also validate that we can automatically select near-optimal strategies based on our I/O cost model.

View Materialization. We validate that (1) the co-clustering strategies that we explored in Section 3 outperform both Lazy and Eager strategies, and (2) neither of the co-clustering strategies dominates the other. We compare the four materialization strategies; our GLAYOUT heuristic page-oriented layout is used in all settings. We report results for both EleFILE and EleHBASE on **CRF** and **LDA**.

As shown in Figure 14(a), for **CRF**, V-CoC dominates other strategies for all settings. We looked into the buffer manager; we found that (1) V-CoC outperforms Lazy (resp. F-CoC) by incurring 75% fewer read misses on **Sp/Sb** (resp. **Sp/Lb**); and (2) V-CoC incurs 20% fewer dirty page writes than Eager. The situation, however, is reversed in **LDA**. Here, F-CoC dominates, as both V-CoC and Eager fail to run **LDA** as they would require 100TB+ of disk space for materialization. Thus, neither co-clustering strategy dominates the other, but one always dominates both Eager and Lazy. The same observation holds for both EleFILE and EleHBASE.

A key difference between these tasks is that the number of variables in most factors in **LDA** is much larger (can be larger than 1,000) than **LR** and **CRF** (fewer than 3). Thus, some pre-materialization strategies are expensive in **LDA**. On the other hand, for **LR** and **CRF**, the time needed to look up factors is the bottleneck (which is optimized by V-CoC).

We then validate that one can automatically select between different materialization strategies by using the cost model shown in Figure 5. For each model and materialization strategy, we estimated the throughput as the inverse of the estimated total amount of I/O. For each materialization strategy, we plot its estimated and actual throughput. Figure 13 shows the results for EleFILE (**Sp/Lb**). We can see that the throughput decreases when the estimated throughput decreases. Based on this cost model, we can select the near-optimal materialization strategy.

Page-oriented Layout. We validate that our heuristic ordering strategy can improve sampling throughput over a random ordering. We focus on the F-CoC materialization strategy here, and report the V-CoC setting in the full version because the results are similar. We compare two page-oriented layout strategies: (1) **Shuffle**, in which we randomly shuffle the factor table, and (2) **Greedy**, which use our greedy heuristic to layout the factor table. As shown in Figure 14(b), **Greedy** dominates **Shuffle** on all data sets in both settings. For **CRF**, the heuristic can achieve two orders of magnitude of speedup. We found that for **Sp/Lb**, heuristic ordering incurs two orders of magnitude fewer page faults than random ordering. These results show that the tradeoff in page-oriented layout impacts the throughput of Gibbs sampling on factor graphs that do not fit in memory.

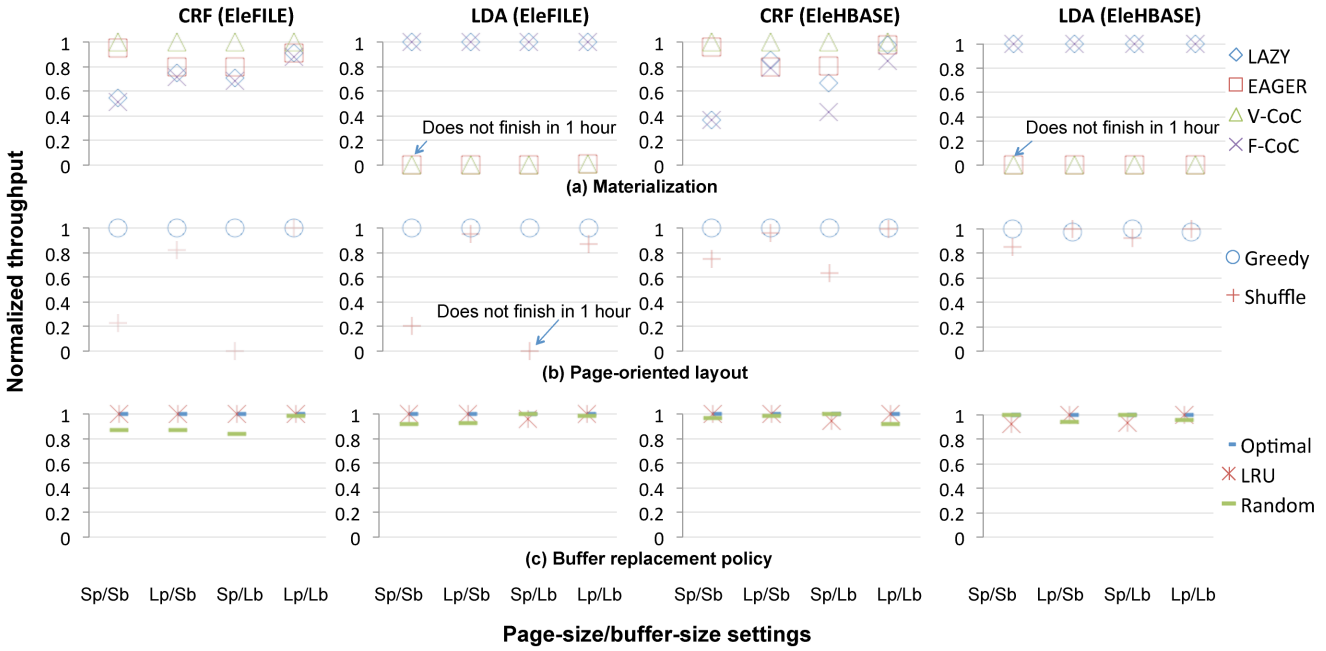

Figure 14: I/O tradeoffs on *Perf* dataset. Recall that Sp (resp. Lp) means *small (resp. large) page size*; Sb (resp. Lb) means *small (resp. large) buffer size*. See Figure 9 for the exact definitions of page and buffer size settings

Buffer-Replacement Policy. We validate that the optimal buffer-replacement strategy can only achieve a marginal improvement of sampling throughput over LRU. We compare three settings: (1) **Optimal**, which uses the optimal caching strategy on tables with random access, (2) **LRU**, which uses LRU on tables with random access; and (3) **Random**, which uses a random replacement policy. We use MRU on tables with sequential scan. We use V-CoC for **LR** and **CRF**, F-CoC for **LDA**, and **Greedy** for all page-oriented layouts because they are the optimal strategies for these tasks. As shown in Figure 14(c), **Optimal** achieves 10% higher throughput than **Random** for **CRF**, and 5% higher throughput than **LRU** for **LDA**. For **CRF**, the access pattern of factors is much like a sequence scan, and therefore **LRU** is near-optimal. As **Optimal** requires non-trivial implementation and is not implemented in existing data processing systems, an engineer may opt against implementing it in a production system.

Discussion. On this small set of popular tasks, our experiments demonstrate that our prototype can achieve competitive (and sometimes better) throughput and scalability than state-of-the-art systems for Gibbs sampling from factor graphs that are larger than main memory. What is interesting to us about this result is that we used a set of generic optimizations inspired by classical database techniques. Of course, in any given problem, there may be problem-specific optimizations that can dramatically improve throughput. For example, in higher-level languages like MCDB or Factorie, there may be additional optimization opportunities. Our goal, however, is not to exhaustively enumerate such optimizations; instead, we view the value of our work as articulating a handful of tradeoffs that may be used to improve the throughput of either specialized or generic systems for Gibbs sampling from factor graphs. These tradeoffs have not been systematically studied in previous work. Also, our techniques are orthogonal to optimizations that choose between various sampling methods to improve runtime performance, e.g., Wang et al. [42]. A second orthogonal direction is distributing the data to many machines; such an approach would allow us to achieve higher scalability and performance. We believe that the optimizations that we describe are still useful in the distributed setting. However, there are new tradeoffs that may need to be considered, e.g., communication and replication tradeoffs are obvious areas to explore.

5. RELATED WORK

We describe related work from the Gibbs sampling literature and the probabilistic databases literature.

Probabilistic graphical models [21] and factor graphs, in particular, are popular frameworks for statistical modeling. Gibbs sampling has been applied to a wide range of probabilistic models and applications [3], e.g., risk models in actuarial science [4], conditional random fields for information extraction [14], latent Dirichlet allocation for topic modeling [16, 17], and Markov logic for text applications [12, 28].

There are several recent general-purpose systems in which a user specifies a factor graph model and the system performs sampling-based inference. For example, Factorie [26] allows a user to specify arbitrary factor functions in Java; OpenBUGS [25] provides an interface for a user to specify arbitrary Bayes nets (which are also factor graphs) and performs Gibbs sampling on top of them; PGibbs [15] performs parallel Gibbs sampling over factor graphs on top of GraphLab [24]. We hope the tradeoffs we describe here add

to this line of work for factor graphs that are larger than main memory.

To deploy sophisticated statistical analysis over increasingly data-intensive applications, there is a push to combine factor graphs and databases [37,44]. However, Sen et al. [37] did not consider Gibbs sampling. Most closely related to our work is Wick et al. [44], who do perform blocked Gibbs sampling and exploit a connection to materialized view maintenance. However, their approach performs the core operation in main memory. Hence, they did not consider the tradeoffs that we discuss in this paper.

Gibbs sampling is often optimized for specific applications. For example, a popular technique for implementing Gibbs sampling for latent Dirichlet allocation [16, 23, 32, 38] is to only maintain the sum of a set of Boolean random variables instead of individual variable assignments. A more aggressive parallelization strategy allows non-atomic updates to the variables to increase throughput (possibly at the expense of sample quality) [23,38]. Our prototype does allow this optimization, but it is orthogonal to our contributions (and we do not use it in our experiments).

One line of work tries to scale up these main-memory systems to support analytics on data sets that are larger than main memory. For example, the Ricardo system [11] integrates R and Hadoop to process terabytes of data stored on HDFS. However, Ricardo spawns R processes on Hadoop worker nodes, and still assumes that statistical analytics happen in main memory on a single node. Our system, however, studies how to scale up Gibbs sampling using both main memory and secondary storage. Systems like Ricardo could take advantage of our results to get higher scalability by improving single-node scalability. The MCDB system [20] is the seminal work of integrating sampling-based approaches into a database. MCDB scales up the sampling process using an RDBMS. However, it assumes that the classic I/O tradeoffs for an RDBMS work in the same way for sampling. Our work revisits these classical tradeoffs, and we hope that our study contributes to this line of work.

There is a rich literature on pushing probabilistic models and algorithms into database systems [1, 5, 10, 19, 29, 31, 35, 37, 42, 44]. Factor graphs [41] are one of the most popular and general formalisms [37, 42, 44]. There are alternative representation frameworks to factor graphs. For example, BayesStore [43] uses Bayes nets; Trio [1, 35], MayBMS [5], and MystiQ [10] use *c*-tables [18]; MCDB [19] uses VG functions; and Markov logic systems such as Tuffy [28] use grounded-clause tables. We focus on factor graphs but speculate that our techniques could apply to other probabilistic frameworks as well. PrDB [37] uses factor graphs and casts SQL queries as inference problems in factor graphs. While PrDB represents factor functions by enumerating all possible outcomes, we observe that such explicit enumeration is inefficient for factors involving hundreds of variables (e.g., in LDA), and so our system follows Factorie [26] and represents factor functions as C++ functions. PrDB could adopt this representation as well.

6. CONCLUSION

Motivated by problems in a variety of governmental and industrial applications, we studied how to make a highly scalable Gibbs sampler. All other things being equal, a Gibbs sampler that produces samples with higher throughput allows us to obtain high quality more quickly. Thus, the central technical challenge that we study is how to create a high-throughput sampler. In contrast to previous approaches that optimize sampling by proposing new algorithms, we examined whether or not traditional database optimization techniques could be used to improve the core operation of Gibbs sampling. In particular, we study the impact of a handful of classical data management tradeoffs on the throughput of Gibbs sampling on factor graphs that are larger than main memory. Our work articulates a handful of tradeoffs that may be used to improve the throughput of such systems, including materialization, page-oriented layout, and buffer-management tradeoffs. After analytically examining the tradeoff space for each technique, we described novel materialization strategies and a greedy page-oriented layout strategy for factor graphs.

Our experimental study found that our techniques allowed us to achieve up to two orders of magnitude improvement in throughput over classical techniques. Using a classical result due to Bélády, we implemented the optimal page-replacement strategy, but found empirically that it offered only modest improvement over traditional techniques. Overall, we saw that by optimizing for only these classical tradeoffs, we were able to construct a prototype system that achieves competitive (and sometimes much better) throughput than prior Gibbs sampling systems for factor graphs. As our optimization techniques are orthogonal to other techniques in the literature, we hope that our techniques will contribute to a wide array of sampling systems that have been developed in recent years.

7. ACKNOWLEDGMENT

We gratefully acknowledge the support of Defense Advanced Research Projects Agency (DARPA) DEFT Program under Air Force Research Laboratory (AFRL) prime contract No. FA8750-13-2-0039, the National Science Foundation EAGER Award under No. EAR-1242902 and CAREER Award under No. IIS-1054009, the Office of Naval Research under awards No. N000141210041 and No. N000141310129, Sloan Research Fellowship, American Family Insurance, and Google. Any opinions, findings, and conclusion or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of DARPA, AFRL, ONR, NSF, or the US government.

8. REFERENCES

[1] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. Nabar, T. Sugihara, and J. Widom. Trio: A system for data, uncertainty, and lineage. In *VLDB*, 2006.

[2] K. Andreev and H. Räcke. Balanced graph partitioning. In *SPAA*, 2004.

[3] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan. An Introduction to MCMC for Machine Learning. *Machine Learning*, 2003.

[4] K. Antonio and J. Beirlant. Actuarial statistics with generalized linear mixed models. In *Insurance: Mathematics and Economics*, 2006.

[5] L. Antova, C. Koch, and D. Olteanu. Query language support for incomplete information in the MayBMS system. In *VLDB*, 2007.

[6] D. Barber. *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2012.

[7] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 1966.

[8] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet allocation. *J. Mach. Learn. Res.*, 2003.

[9] F. Chen, X. Feng, C. Re, and M. Wang. Optimizing statistical information extraction programs over evolving text. In *ICDE*, 2012.

[10] N. Dalvi, C. Re, and D. Suciu. Queries and materialized views on probabilistic databases. *JCSS*, 2011.

[11] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: Integrating R and Hadoop. In *SIGMOD*, 2010.

[12] P. Domingos and D. Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence*. Morgan and Claypool Publishers, 2009.

[13] N. Fenton and M. Neil. Combining evidence in risk analysis using Bayesian networks. *Agena*, 2004.

[14] J. Finkel, T. Grenager, and C. Manning. Incorporating non-local information into information extraction systems by Gibbs sampling. In *ACL*, 2005.

[15] J. Gonzalez, Y. Low, A. Gretton, and C. Guestrin. Parallel Gibbs sampling: From colored fields to thin junction trees. In *AISTATS*, 2011.

[16] T. L. Griffiths and M. Steyvers. Finding scientific topics. *PNAS*, 2004.

[17] J. M. e. a. Hellerstein. The MADlib analytics library: or MAD skills, the SQL. *PVLDB*, 2012.

[18] T. Imieliński and W. Lipski, Jr. Incomplete information in relational databases. *J. ACM*, 1984.

[19] R. Jampani, F. Xu, M. Wu, L. Perez, C. Jermaine, and P. J. Haas. The Monte Carlo database system: Stochastic analysis close to the data. *TODS*, 2011.

[20] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. M. Jermaine, and P. J. Haas. MCDB: A Monte Carlo approach to managing uncertain data. In *SIGMOD*, 2008.

[21] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[22] J. D. Lafferty, A. McCallum, and F. C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML*, 2001.

[23] Z. Liu, Y. Zhang, E. Y. Chang, and M. Sun. PLDA+: Parallel latent Dirichlet allocation with data placement and pipeline processing. *ACM TIST.*, 2011.

[24] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *UAI*, 2010.

[25] D. Lunn, D. Spiegelhalter, A. Thomas, and N. Best. The BUGS project: Evolution, critique and future directions. *Statistics in Medicine*, 2009.

[26] A. McCallum, K. Schultz, and S. Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. In *NIPS*, 2009.

[27] D. Newman, P. Smyth, and M. Steyvers. Scalable parallel topic models. *Journal of Intelligence Community Research and Development*, 2006.

[28] F. Niu, C. Ré, A. Doan, and J. W. Shavlik. Tuffy: Scaling up statistical inference in Markov logic networks using an RDBMS. *PVLDB*, 2011.

[29] D. Olteanu, J. Huang, and C. Koch. Sprout: Lazy vs. eager query plans for tuple-independent probabilistic databases. In *ICDE*, 2009.

[30] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1st edition, Sept. 1988.

[31] L. Peng, Y. Diao, and A. Liu. Optimizing probabilistic query processing on continuous uncertain data. *PVLDB*, 2011.

[32] I. Porteous, D. Newman, A. Ihler, A. Asuncion, P. Smyth, and M. Welling. Fast collapsed Gibbs sampling for latent Dirichlet allocation. In *KDD*, 2008.

[33] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. Osborne/McGraw-Hill, 2000.

[34] C. P. Robert and G. Casella. *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Springer-Verlag New York, Inc., 2005.

[35] A. D. Sarma, O. Benjelloun, A. Halevy, S. Nabar, and J. Widom. Representing uncertain data: Models, properties, and algorithms. *The VLDB Journal*, 2009.

[36] S. Satpal, S. Bhadra, S. Sellamanickam, R. Rastogi, and P. Sen. Web information extraction using Markov logic networks. In *WWW*, 2011.

[37] P. Sen, A. Deshpande, and L. Getoor. PrDB: Managing and exploiting rich correlations in probabilistic databases. *The VLDB Journal*, 2009.

[38] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. *PVLDB*, 2010.

[39] J. R. Sobehart, R. Stein, V. Mikityanskaya, and L. Li. Moody's public firm risk model: A hybrid approach to modeling short term default risk. *Moody's Investors Service, Global Credit Research, Rating Methodology*, 2000.

[40] C. Sutton and A. Mccallum. Collective segmentation and labeling of distant entities in information extraction. Technical Report #04-49, U. Mass, 2004.

[41] M. J. Wainwright and M. I. Jordan. Graphical models, exponential families, and variational inference. *Found. Trends Mach. Learn.*, 2008.

[42] D. Z. Wang, M. J. Franklin, M. Garofalakis, J. M. Hellerstein, and M. L. Wick. Hybrid in-database inference for declarative information extraction. In *SIGMOD*, 2011.

[43] D. Z. Wang, E. Michelakis, M. Garofalakis, and J. M. Hellerstein. BayesStore: Managing large, uncertain data repositories with probabilistic graphical models. *PVLDB*, 2008.

[44] M. Wick, A. McCallum, and G. Miklau. Scalable probabilistic databases with factor graphs and mcmc. *PVLDB*, 2010.

[45] T. Xu and A. T. Ihler. Multicore Gibbs sampling in dense, unstructured graphs. *AISTATS*, 2011.

[46] J. Zhu, Z. Nie, X. Liu, B. Zhang, and J.-R. Wen. StatSnowball: A statistical approach to extracting entity relationships. In *WWW*, 2009.