Divergent Physical Design Tuning for Replicated Databases

Mariano P. Consens
Univ. of Toronto
consens@cs.toronto.edu

Kleoni Ioannidou
UC Santa Cruz
kleoni@cs.ucsc.edu

Jeff LeFevre
UC Santa Cruz
jlefevre@cs.ucsc.edu

Neoklis Polyzotis
UC Santa Cruz
alkis@cs.ucsc.edu

ABSTRACT

We introduce *divergent designs* as a novel tuning paradigm for database systems that employ replication. A divergent design installs a different physical configuration (e.g., indexes and materialized views) with each database replica, specializing replicas for different subsets of the workload. At runtime, queries are routed to the subset of the replicas configured to yield the most efficient execution plans. When compared to uniformly designed replicas, divergent replicas can potentially execute their subset of the queries significantly faster, and their physical configurations could be initialized and maintained (updated) in less time. However, the specialization of divergent replicas limits the ability to load-balance the workload at runtime.

We formalize the divergent design problem, characterize the properties of good designs, and analyze the complexity of identifying the optimal divergent design. Our paradigm captures the trade-off between load balancing among all n replicas vs. load balancing among $m \leq n$ specialized replicas. We develop an effective algorithm (leveraging single-node-tuning functionality) to compute good divergent designs for all the points of this trade-off. Experimental results validate the effectiveness of the algorithm and demonstrate that divergent designs can substantially improve workload performance.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design

General Terms

Performance

Keywords

physical design tuning, divergence, replicated databases

1. INTRODUCTION

Data replication is an ubiquitous feature in distributed database systems. In a nutshell, the system maintains several copies (or replicas) of the data at different nodes, and ensures that they remain synchronized when updates are applied to the database. Replication may occur at different levels of granularity, from subsets of tuples to a complete database. In all cases, the main motivation behind replication is to cope with node failures and ensure high availability, but quite often replication is also used to enable the parallel servicing of a workload in a load-balanced fashion.

In this work, we introduce the paradigm of a *divergent design*, which leverages replication for the purpose of tuning the database system more effectively. Specifically, a divergent design installs a different physical configuration (e.g., indexes and materialized views) with each replica in order to specialize it for a subset of the workload. In this fashion, a query statement in the incoming workload is routed to the replica that can evaluate it most efficiently. Our proposed approach is in contrast to the current state of the practice, where each replica has the exact same physical configuration (a *uniform design*). By allowing the configuration of each replica to diverge, a divergent design utilizes the aggregate storage in the system more effectively (i.e., more indexes and materialized views are built overall) and consequently enables higher benefits for more queries in the workload together with faster initialization and maintenance of replicas.

Several real-world systems employ replication and can thus benefit directly from divergent designs. Database systems that run in the cloud using a Database-as-a-Service provider are one example. These providers typically employ commodity hardware to run the database system and rely on replication to cope with failures as well as achieve scalability. Two such examples are Microsoft SQL Azure [2], which employs 3-way replication, and Amazon's Relational Database Service [1], where up to 5 read replicas of MySQL databases can be launched on-demand to provide scalability. These systems fully replicate the database on several nodes, thus allowing a query to be evaluated on any replica. In both cases, a divergent design will allow replicas to specialize for subsets of the workload and thus improve performance of query processing. A similar case can also be made for parallel database systems which employ replication for high availability, failure recovery, and performance e.g., the recently announced Teradata Unity feature. In addition, divergent designs can be particularly beneficial in the context of ad-hoc data analytics. For instance, a data scientist may procure several machines from a private or public cloud (e.g., Amazon EC2), spin up several database instances for the same data set, and then use divergent designs to efficiently evaluate a heavy analytics workload in parallel. In all these examples, using a divergent design does not require any changes to the underlying query processor. It is only a matter of installing a different configuration on each replica and taking care of routing queries to the appropriate replicas.

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.

The focus of our work is the problem of *divergent design tuning*, i.e., how to compute a beneficial divergent design for a system that employs database replication. The problem involves numerous technical challenges. Choosing a good divergent design requires reasoning about possible subsets of the workload and how they can benefit from possible configurations, which already leads to a doubly-exponential space of solutions. In particular, divergent design tuning can be viewed as a generalization of physical design tuning for a single node, which is already known to be a hard problem [5]. Another source of complication is the tension between best-case performance and load balancing. Specifically, the specialization of replicas to different subsets of the workload may cause a query to have really good performance on one replica but really bad performance everywhere else. In turn, this variability can result in load imbalance and bad overall performance, e.g., in the case where a query cannot be routed to its "best" replica. Our work shows that it is non-trivial to determine a good trade-off point between specialization and load balancing.

There is a long line of studies on physical design tuning, however no previous work examines the concept of divergent designs that we introduce in our work. Several studies investigate the problem of tuning the configuration of a single system [3, 4, 10], but they do not examine the interplay with replication. In fact, the current state of the practice is to apply these techniques to tune the configuration of a single replica, and then to copy the resulting configuration on every other replica. The concept of fractured mirrors [12] is similar in spirit to divergent designs, except that it assumes exactly two replicas of the database that are stored in row- and column-based format respectively, and also requires a modified query processor. In contrast, a divergent design installs a "richer" configuration on each replica based on the data structures (e.g., indexes, materialized views, row- and column-based storage) that are already supported by the query processor. (A more detailed review of related work appears in Section 7.)

Contributions. The technical contributions of our work can be summarized as follows.

- We formally introduce the concept of a divergent design and define the problem of divergent design tuning (Section 2). Our formalization casts the problem as computing a partition of the workload across replicas, and then using existing tools (e.g., DB2's Design Advisor, or the Index Tuning Wizard of MS SQL Server) to compute the corresponding configurations. In this fashion, a divergent design yields configurations that are readily tailored to the capabilities of the underlying query processor. Our definition also models the tension between divergence and load balancing, by requiring each query to have a least number of "good" replicas.

- We analyze the complexity of the problem and the properties of good divergent designs (Section 3). Not surprisingly, our analysis reveals that finding the optimal divergent design is an NP-Hard problem. At the same time, we identify an intuitive property that characterizes good divergent designs, and which we use in the development of our tuning algorithm. We prove that the current state of the practice (a uniform design replicating the single-system configuration to all replicas) can be suboptimal. Furthermore, we show that that there is a big performance gap between good and bad designs, and that there are few good designs (hence, chances are against an application design been optimal).

- We propose an efficient algorithm to compute effective divergent designs (Section 4). The algorithm computes an initial partition of the workload and then refines it in an iterative fashion. We formally show that each iteration leads to a better partition and also that the generated designs have provably good properties.

- We present an extensive experimental study using a commercial DBMS to evaluate the potential of divergent designs and also the effectiveness of our tuning algorithm (Section 5). The results demonstrate that our tuning algorithm runs efficiently and yields divergent designs that have several benefits: the available space budget for physical configurations is utilized more effectively; the time to build the configuration of each replica drops significantly as the configuration size increases; update statements are executed faster; and, query performance improves significantly, without compromising the ability to load-balance the system.

We also discuss the implementation of divergent designs in an actual DBMS (Section 6). Existing systems can already reap some benefits from divergent designs. However, to realize the full potential of divergent design tuning, it becomes necessary to modify the policies for query routing and load balancing.

2. DIVERGENT DESIGN TUNING: PROBLEM STATEMENT

Preliminaries. We first review some basic concepts from the conventional problem of physical design tuning over a single database. The problem of divergent design tuning extends these concepts to a parallel environment where the database is replicated.

In conventional physical design tuning, we are given a representative workload W and a space budget b, and the goal is to compute a configuration that minimizes the cost to evaluate W and fits within the space budget b. The computed configuration typically comprises materialized views and (primary or secondary) indexes over tables and views. Formally, let $W = Q \cup U$ where Q is the *query workload* and U is the *update workload*. Since W plays the role of a representative workload, it is common to provide a weight function $f : W \rightarrow \Re$, such that $f(x)$ corresponds to the importance of query or update statement x in W. W and f are typically specified by the administrator or they can be obtained automatically, e.g., by mining the query logs of the database system.

We use $cost(x, I)$ to denote the cost of evaluating (query or update) statement x assuming that I is the configuration of the system. The cost function can be evaluated efficiently in modern systems (i,e., without materializing I) using a *what-if optimizer* [7]. Given a configuration I, we can define the cost of evaluating W as follows:

$$SingleCost(W, f, I) = \sum_{q \in Q} f(q) cost(q, I) + \sum_{u \in U} f(u) cost(u, I)$$

Building on these definitions, the physical design tuning problem is defined as computing the configuration I such that I has size not greater than b and $SingleCost(W, f, I)$ is minimal. Physical design tuning has been the focus of a long line of research studies [3, 4, 10], and most commercial database management systems come with a *configuration advisor* tool that automates the tuning process [6, 9, 18]. We model a configuration advisor as a function $DBAdv$ that takes as input the workload W, the weight function f, and the space budget b and outputs a configuration I, i.e., $I = DBAdv(W, f, b)$. Since computing the optimal configuration is an NP-Hard problem, the advisor relies on heuristics to output a good solution that is hopefully close to optimal.

Divergent Physical Design Tuning. We extend the previous concepts to a system where the database is replicated across n machines. We consider the cases described in Section 1, where each replica $i \in [1, n]$ holds a full copy of the database. Moreover, we assume that replicas have the same space budget b for their configuration. Our techniques are readily extensible to the case where the unit of replication is a partition of the database, or when replicas may have different space budgets.

In this particular setting, each query statement q can be evaluated by any replica in the system. (Of course, the performance of query evaluation may be different depending on which replica is chosen.) On the other hand, each update statement u has to be evaluated at each replica, in order to ensure consistency. The system may use different strategies to ensure synchronization for updates, e.g., eager vs lazy, but this choice is completely orthogonal to our techniques.

The high-level idea of a divergent physical design is to allow each replica to have a different physical design, tailored to a particular subset of the workload. Formally, let $\{Q_1, \ldots, Q_n\}$ denote a partition of the query workload Q to n (potentially overlapping) subsets. The partition of Q induces a partition of W in corresponding subsets $W_1, \ldots, W_n$, that we call *sub-partitions* such that $W_i = Q_i \cup U$. We use $partitions(W)$ to denote the set of such partitions. Essentially, W_i represents the subset of the workload for which replica i is specialized. Note that each W_i contains U, since all updates have to be applied to all replicas. We couple the partition $\{W_1, \ldots, W_n\}$ with a set of corresponding weight functions $\{f_1, \ldots, f_n\}$, where $f_i()$ sets the weights of queries and updates in W_i. (We will impose certain restrictions on f_i later.) In what follows, we use $p = \{(W_1, f_1), \ldots, (W_n, f_n)\}$ to denote the combined information of workload subsets and weight functions. Note that the *uniform design* $p_{unif} = \{(W, f), \ldots, (W, f)\}$ captures the current practice in real systems, where each replica is equipped with the same configuration $I_{unif} = DBAdv(W, f, b)$ that is computed based on the complete workload.

DEFINITION 2.1 (DIVERGENT DESIGN). *Given a workload $W = Q \cup U$, a weight function f, and a number of replicas n, a divergent design corresponds to a set $p = \{(W_1, f_1), \ldots, (W_n, f_n)\}$, such that:*

- *$W_i = Q_i \cup U$ for all $i \in [1, n]$, where $Q_1 \cup \cdots \cup Q_n = Q$.*
- *$f_i(u) = f(u)$ for all $u \in U$.*
- *$\sum_{i \in [1,n]} f_i(q) = f(q)$.*

The configuration of each replica is computed as $I_i = DBAdv(W_i, f_i, b)$.

Our definition places three constraints on p. The first one corresponds to our earlier observation that each W_i contains all the updates. The second and third constraints regulate the weight functions $f_i()$. Specifically, $f_i(u) = f(u)$ preserves the importance of an update *on each replica*, whereas $\sum_i f_i(q) = f(q)$ preserves the weight of a query *across all replicas*. These properties are aligned with the setting that we consider: each occurrence of u has to be routed to every replica, whereas each occurrence of a query q is routed to one replica. The definition also states that each replica will be tuned with a configuration I_i that results from invoking $DBAdv()$ on the corresponding subset W_i and weight vector f_i.

An implication of our definitions is that we essentially view the function $DBAdv()$ as a black box. We chose this modeling for two reasons. First, we reduce the problem of divergent design tuning to that of finding the right partition of the workload and corresponding weight functions that maximize a suitable performance metric (to be detailed shortly). Second, we can leverage the existing literature on automated tuning techniques, and in particular the existing advisor tools, in order to compute the configuration of each replica.

To model the performance of a divergent design p we could extend the $SingleCost$ metric, where each query q in W is now evaluated by the replica with the best physical design for q. Formally, let $I_1, \ldots, I_n$ denote the replica configurations corresponding to p. We define $J_1^q, \ldots, J_n^q$ as the permutation of $I_1, \ldots, I_n$ such that $cost(q, J_1^q) \leq cost(q, J_2^q) \leq \cdots \leq cost(q, J_n^q)$, i.e., J_j^q is the physical design of the j-th best replica for q. Hence, the best-case total evaluation cost for W can be expressed as follows:

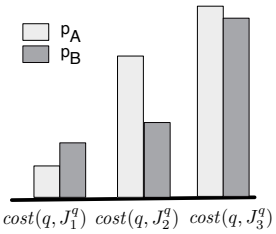

Figure 1: An illustration of the trade-off between load balancing and performance.

$$BestCost(p) = \sum_{q \in Q} f(q) cost(q, J_1^q) + \sum_{u \in U} \sum_{i \in [1,n]} f(u) cost(u, I_i)$$

This metric reflects the total work performed by the system when each query is routed to its best replica and each update is routed to all replicas.

The problem with the previous metric is that it ignores completely the issues of load-balancing and parallel processing for queries. Specifically, the metric does not guard against the case where $cost(q, J_1^q) << cost(q, J_2^q)$ and hence q performs badly on every replica except the one with configuration J_1^q. A schematic representation is given in Figure 1, which depicts $cost(q, J_j^q)$ for a system with $n = 3$ replicas. Design p_A ensures a low cost for the best replica of q, but a much higher cost for other replicas. On the other hand, design p_B ensures that two replicas can evaluate q with low cost, albeit higher than the best-case replica of p_A. In a sense, design p_B trades off best-case performance for flexibility in routing queries. The uniform design p_{unif} stands at the extreme point of this trade-off, since every query has the same cost on every replica.

To address load-balancing in our problem statement, we introduce a load-balancing factor $m \leq n$ that specifies the number of "low-cost" replicas in p for each query in Q. We expect the DBA to set m based on the desired behavior of the system. If parallel processing and load balancing are not important, then $m = 1$ will optimize the choice of p for the best-case cost (essentially, the previous metric). On the other hand, $m = n$ will lead to a design p such that q has similar performance on all replicas. Values of m between these two extremes will trade off best-case performance for the ability to load balance. We incorporate factor m in the performance metric as follows:

$$\begin{aligned} TotalCost(p, m) \quad = \quad & \sum_{q \in Q} \sum_{j \in [1,m]} \frac{f(q)}{m} cost(q, J_j^q) + \\ & \sum_{u \in U} \sum_{i \in [1,n]} f(u) cost(u, I_i) \end{aligned}$$

The second term is simply the cost to push updates to all the replicas. The first term reflects the total work to evaluate the workload, assuming that the cost of each query q is "distributed" across the m replicas with the best physical designs for q. Hence, this metric favors a divergent design that ensures at least m good options for each query. Our proposed metric is a generalization of the cost

metric for conventional physical design tuning, to the case where m replicas equally share the cost of each query. Returning to the example of Figure 1, design p_B outperforms p_A for $m = 2$, but p_A is better for $m = 1$.

We now formulate the problem that we tackle in this paper, namely computing the optimal divergent design.

DEFINITION 2.2 (OPTIMAL DIVERGENT DESIGN). *Given a workload W, a weight function f, a number of replicas n and a load-balancing factor $m \leq n$, compute the divergent design p that minimizes $TotalCost(p, m)$.*

We can also formulate an alternative problem statement, where we optimize m based on a desired trade-off between best-case performance and load balancing.

DEFINITION 2.3 (OPTIMAL m AND DESIGN). *Given a workload W, a weight function f, a number of replicas n, a threshold $\tau \in (0, 1]$ and the uniform design $p_{unif} = \{(W, f), \ldots, (W, f)\}$, compute the maximal m and corresponding divergent design p such that $TotalCost(p, m) \leq \tau\, TotalCost(p_{unif}, n)$.*

In this case, we seek to find the divergent design that offers the most opportunities for load balancing and improves on the baseline uniform design at least by a factor of τ. In the extended version of the paper, we prove that any PTIME algorithm to solve the first problem yields a PTIME algorithm to solve the second problem, by using binary search to identify the maximal value for m. Hence, from this point onward, we focus on the first problem.

For purposes that will become clear in the next section, we distinguish the class of *l-balanced divergent designs*. The formal definition follows.

DEFINITION 2.4 (l-BALANCED DESIGN). *A divergent design $p = \{(W_1, f_1), \ldots, (W_n, f_n)\}$ is called l-balanced iff the following two conditions hold:*

- *Each $q \in W$ appears in exactly l sub-partitions of p, i.e., $\{i \mid i \in [1, n] \wedge q \in W_i\}$ has exactly l elements for any $q \in W$.*
- *$f_i(q) = f(q)/l$ if $q \in W_i$ and $f_i(q) = 0$ otherwise, for any $q \in W$ and $i \in [1, n]$.*

In an l-balanced design p, each query q directly affects the configuration of exactly l replicas. Moreover, the original weight of q is uniformly distributed among the l sub-partitions that contain q. A special case of an l-balanced design is the *fully-balanced design* $p_{\text{FB}} = \{(W, f_{\text{FB}}), \ldots, (W, f_{\text{FB}})\}$ where $l = n$, i.e., each query q appears in every sub-partition with weight $f_{\text{FB}}(q) = f(q)/n$. Similar to the uniform design, the fully-balanced design installs the same configuration $I_{\text{FB}} = DBAdv(W, f_{\text{FB}}, b)$ in each replica. The difference is that queries have their weights deflated by n, which intuitively reflects the fact that q can be evaluated by any of the n replicas with equal cost $cost(q, I_{\text{FB}})$. The counter-part to the fully-balanced design p_{FB} is a 1-balanced design p, where each query q directly affects the configuration of exactly one replica. Note that *application designs* (when modeling each query as occurring in only one application) are 1-balanced designs with the disjoint partition determined by the given applications.

3. PROBLEM ANALYSIS

Having defined the problem of divergent design tuning, our next step is to develop some intuition about the space of possible solutions. Our approach is two-pronged. First, we present a theoretical analysis on the complexity of the problem and the properties of good divergent designs. We couple these theoretical results with an exhaustive investigation of the space of divergent designs for a small-scale instance of the problem. Overall, the theoretical results and our empirical observations demonstrate the challenges behind the problem: there is a vast search space of divergent designs; identifying the optimal design is computationally hard; there exists a very big performance gap between good and bad designs; and, there are relatively few good designs. These insights contribute directly to the design of the tuning algorithm presented in the next section.

3.1 Theoretical Analysis

For the purpose of our theoretical analysis, we assume that $DBAdv(W, f, b)$ returns the *optimal* configuration for any W, f and b. This assumption allows us to analyze the properties of divergent design tuning without having to model the imperfections of the advisor. We stress that the tuning algorithm we present in Section 4 *does not* rely on this assumption.

First, we investigate the computational complexity of identifying the optimal divergent design.

THEOREM 3.1. *The optimal divergent design cannot be computed in polynomial time unless $P = NP$.*

The proof shows that the decision problem of divergent design tuning is NP-Hard, through a reduction from the NP-Complete subset sum problem. Therefore, instead of looking for an optimal algorithm to solve the problem, our goal is to develop an efficient algorithm that computes good divergent designs. In this direction, we analyze further the space of possible divergent designs in order to build some intuition on the properties of good solutions.

Recall that the fully-balanced design p_{FB} installs the same configuration in all replicas, computed based on the complete workload W and a modified weight function $f_{\text{FB}}(q) = f(q)/n$. The intuition behind p_{FB} is that each query q can be evaluated by any replica with the same cost, and hence the original weight $f(q)$ is equally spread across all replicas. This intuition makes the fully-balanced design a natural choice for $m = n$, and in fact we prove that it is an optimal solution.

THEOREM 3.2. *The fully-balanced design p_{FB} is optimal if $m = n$, i.e., $TotalCost(p_{FB}, n) \leq TotalCost(p, n)$ for any other design p.*

Our next step is to consider the other extreme point of $m = 1$, where we care for best-case performance. A 1-balanced design is a natural choice for $m = 1$, since it naturally associates each query with a single replica. However, proving optimality is far from trivial, given that there are many such designs and also because the best cost for q may not come from the configuration that has considered q. Still, we are able to show an important result: the fully-balanced design cannot beat *any* 1-balanced design for $m = 1$. In fact, we prove the following generalization:

THEOREM 3.3. *$TotalCost(p, m) \leq TotalCost(p_{FB}, m)$ for any $m \in [1, n]$ and any m-balanced design p.*

This result reflects the tension between load-balancing and performance. The fully-balanced design offers the greatest flexibility for load-balancing (and is in fact optimal for $m = n$), since each query q has the same cost on each replica. On the other hand, an m-balanced design tries to specialize m replicas per query, which reduces the flexibility for load-balancing but improves the cost of workload evaluation.

The previous two theorems lead us to the following intuition about the properties of a "good" divergent design p: *The overlap between sub-partitions should increase as m increases.* Indeed, the fully-balanced design p_{FB}, whose sub-partitions W_i overlap completely, is optimal for $m = n$. Conversely, for any $m < n$, p_{FB} cannot beat any m-balanced design where each q appears in exactly m sub-partitions. These observations play a crucial role in the divergent design algorithm that we describe in the next section.

We conclude our analysis by comparing divergent designs against the current practice in real systems, which is the *uniform design*. We expect p_{unif} to be mostly relevant for $m = n$, and hence a natural starting point for our analysis is to compare p_{unif} to the optimal design p_{FB}. We prove below that we should always prefer the fully balanced design compared to the uniform design for any value of the load balancing factor.

THEOREM 3.4. *The uniform design cannot outperform the fully balanced design for any load balancing factor, i.e.,* $TotalCost(p_{unif}, m) \geq TotalCost(p_{FB}, m)$ *for any* $m \in [1, n]$

Intuitively, this result can be explained by the asymmetry between queries and updates. Recall that a query q can be evaluated by *any single replica*, whereas an update u has to be evaluated by *all n replicas*. Hence, the contribution of queries to the total work metric decreases by a factor of n compared to the updates. The fully balanced design p_{FB} uses $f_{\text{FB}}(q) = f(q)/n$ for query weights, and therefore the configuration $I_{\text{FB}} = DBAdv(W, f_{\text{FB}}, b)$ takes directly into account this decrease in contribution. On the other hand, p_{unif} ignores the deflation of query importance, as it uses the unmodified query weights. The resulting configuration $I_{unif} = DBAdv(W, f, b)$ is basically oblivious to the increased importance of updates.

We have already seen that p_{FB} cannot lose to p_{unif} (Theorem 3.4), and by Theorem 3.3, we conclude that given a fixed value for m, the uniform design p_{unif} cannot improve on *any* m-balanced design p.

COROLLARY 3.5. $TotalCost(p_{unif}, m) \geq TotalCost(p, m)$ *for any* $m \in [1, n]$ *and any m-balanced design p.*

This general result provides even stronger evidence in favor of substituting p_{unif} (again, the current state of the practice) with a divergent design. Note that *any* m-balanced design has the potential to improve on p_{unif}. The reason can be traced again to the asymmetry between queries and updates, but also to the semantics of the load-balancing factor m. Recall that m specifies that each query q should have m replicas on which it has a low execution cost. An m-balanced design p attempts to achieve this goal by placing q in exactly m sub-partitions and thus having q affect the design of m replicas. This placement still affords some specialization per replica, since each W_i contains a subset of the queries. In contrast, p_{unif} corresponds to a single configuration I_{unif} that is tuned for the whole workload, which means that no replica is specialized. Moreover, p_{unif} may over-provision in terms of load balancing, since each query has the same cost on all $n \geq m$ replicas. These two factors contribute to the sub-optimality of p_{unif} compared to an m-balanced design.

3.2 Empirical Analysis

The second part of our analysis is empirical, and aims to provide further insights on the space of good divergent designs.

We conduct a small-scale empirical study of l-balanced designs with a workload of five queries from the TPC-DS benchmark [15], using IBM DB2 as the underlying DBMS. The details of the experimental setup are given in Section 5. We focused on balanced designs in order to contain the scope of the study and also due to the nice theoretical properties of these designs. We handpicked the queries and tuned the available space budget to simulate the following realistic scenario: different subsets of queries can benefit from similar indexes, but the indexes for different subsets are too large to fit in a single configuration. This is representative of what we expect to see in practice. Intuitively, a good divergent design will specialize a different replica for each subset.

We set $n = 3$ and then exhaustively enumerate all possible l-balanced designs, for $l \in [1, 3]$. Overall, this yielded 486 distinct designs. For each design p, we computed the corresponding configurations I_1, I_2, and I_3 by invoking the DB2 Design Advisor on each sub-partition W_i. Finally, we used DB2's what-if query optimizer to compute $TotalCost(p, m)$ for $m = 1$ and $m = 2$. (We do not consider $m = 3$ since we already know that p_{FB} is optimal.)

We performed the whole computation on Amazon EC2 using ten separate IBM DB2 instances so that we could parallelize the process. Overall, it required many hours of computation time in order to fully explore the search space. The main overhead was the cost of invoking DB2's Design Advisor (several seconds) to compute the physical design for each sub-partition W_i, even with our toy workload. Since an exhaustive search will be impractical for real workloads with tens of queries, any reasonable divergent design tuning algorithm should explore only a small part of the search space.

The results revealed several interesting properties for the space of balanced designs for this particular setup. First, we observed that the optimal design for a fixed value of m is an m-balanced design, for $m \in [1, 3]$. Intuitively, an l-balanced design with $l > m$ provides more flexibility for load balancing than specified by m, and hence misses the opportunity to tighten the specialization of replicas. On the other hand, $l < m$ leads to over-specialization and hence to imbalances among the first m replicas for each query. Based on these empirical observations, we can formulate the following conjecture.

CONJECTURE 3.6. *Let p_m denote an m-balanced divergent design and p_l denote an l-balanced design, for $l \neq m$. Then, it holds that* $\min_{p_m} TotalCost(p_m, m) \leq \min_{p_l} TotalCost(p_l, m)$.

As a next step in our analysis, we examine the distribution statistics of the $TotalCost()$ metric within each class of m-balanced designs, for $m = 1, 2$. In each case, there are 6 designs, out of the 243 designs in the class, which achieve the minimal value for $TotalCost()$. Table 1 further shows the distribution of the $TotalCost()$ values within each class in relation to these optimal designs. For instance, for $m = 1$, the 25-th percentile (that is, the $TotalCost$ value that bounds 25% of the designs) is $\times 2.54$ worse than the optimal cost within the class. Overall, the results show that non-optimal designs have significantly higher cost, and hence a random selection is not likely to yield a good solution to the divergent tuning problem.

It is also interesting to examine the cost of p_{unif} with respect to the optimal in each class. Specifically, $TotalCost(p_{unif}, 1)$ is $\times 26.8$ higher than the optimal cost for the class of 1-balanced designs. The same ratio for $m = 2$ is $\times 5.42$. Hence, the current

	25-th Perc.	Median	75-th Perc.	Max
$m = 1$	$\times 2.54$	$\times 4.47$	$\times 25.1$	$\times 26.8$
$m = 2$	$\times 2.98$	$\times 5.07$	$\times 5.24$	$\times 5.42$

Table 1: Distribution statistics for divergent design costs in the exhaustive experiment. For designs in each cost percentile, we show the ratio of $TotalCost$ to the optimal design cost.

practice of employing a uniform design may lead to a huge loss in performance compared to a divergent design.

4. THE DIVGDESIGN ALGORITHM FOR DIVERGENT DESIGN TUNING

In this section, we present an efficient algorithm called DIVGDESIGN that computes effective divergent designs. We first provide an overview of the key ideas behind the algorithm, and then present the detailed pseudocode. We conclude with a theoretical analysis that demonstrates the nice theoretical properties of DIVGDESIGN.

4.1 Overview of Our Approach

The first design choice behind DIVGDESIGN is to output m-balanced designs, for the value of m in the problem specification. This choice is justified by the theoretical evidence for the nice properties of such designs: the m-balanced design for $m = n$ is optimal; and any m-balanced design has the potential to improve on the baseline p_{unif} partition. The key challenge therefore is to identify an m-balanced design p such that $TotalCost(p, m)$ is small.

Intuitively, a good design should have the property that $J_1^q, \ldots, J_m^q$ (the m best configurations for q among $I_1, \ldots, I_n$) correspond to the configurations of the sub-partitions that contain q. Otherwise, this implies that q affects directly the computation of some configuration $I_i = DBAdv(W_i, f_i, b)$ (by being part of the workload W_i), without however using I_i in the $TotalCost$ metric. In that sense, it is better to take q out of W_i, in order to allow I_i to be tuned even further for the remaining statements in W_i.

Enforcing the aforementioned property is quite challenging, due to the black-box nature of function $DBAdv()$. Essentially, it is infeasible to partition queries based on some analysis of their features, given that we do not make any assumptions about the implementation of $DBAdv()$. Instead, we have to rely on a trial-and-error approach to identify a good partition p, but with a smart strategy that avoids enumerating an exponential number of partitions.

The proposed DIVGDESIGN algorithm employs a strategy that is inspired by the well-known k-means clustering algorithm. The latter is designed to solve clustering problems that employ a black-box distance function. The idea is simple and yet quite effective: start with a random selection of points as the cluster medoids, assign the remaining points to clusters based on the closest medoid, identify the new medoid per cluster, and then iterate in the same fashion until the clustering converges to a stable state. DIVGDESIGN works in a similar fashion. First, it computes a random partition of the workload $\{W_1^0, \ldots, W_n^0\}$ and the corresponding configurations $I_i^0 = DBAdv(W_i^0, f_i^0, b)$, for $i \in [1, n]$. Subsequently, the algorithm computes $cost(q, I_i^0)$ for all $q \in Q$ and $i \in [1, n]$, and "moves" each query q to the sub-partitions that correspond to the lowest evaluation costs. This process yields a new partition $\{W_1^1, \ldots, W_n^1\}$, which is used to compute new configurations $I_1^1, \ldots, I_n^1$ and another assignment of queries to sub-partitions. This iterative process continues until there is convergence.

Figure 2 illustrates an iteration of DIVGDESIGN with a simple example. Assume that we have $n = 2$ replicas, $m = 1$ and the workload contains three queries q_1, q_2, q_3 and no updates. The algorithm starts with an initial random partition $p = \{W_1^0, W_2^0\}$, shown in Figure 2(a). We show the partition in the form of a matrix, where a cell (i, q) is checked if q appears in W_i^0. This partition yields configurations $I_1^0 = DBAdv(W_1^0, f_1^0, b)$ and $I_2^0 = DBAdv(W_2^0, f_2^0, b)$, which are computed by invoking the configuration advisor on the corresponding sub-partitions. Subsequently, the algorithm computes the execution cost of each query under these configurations. An example of resulting costs are shown in

	q_1	q_2	q_3
W_1^0	✓	✓	
W_2^0			✓

(a)

	q_1	q_2	q_3
I_1^0	**10**	20	10
I_2^0	20	**10**	**5**

(b)

	q_1	q_2	q_3
W_1^1	✓		
W_2^1		✓	✓

(c)

Figure 2: An example of running the DIVGDESIGN algorithm for $m = 1$, $n = 2$ and a workload of three queries: (a) Initial design, (b) Query costs under the corresponding configurations (minimum costs are indicated with bold); (c) Refined designs.

Function DIVGDESIGN(W, n, m, b)
Input: Workload W; number of replicas n; load-balancing factor m; space budget b.
Output: A divergent design p that is m-balanced.
Knobs: Improvement threshold ϵ; Max number of iterations $IterMax$

1 Pick a random m-balanced design $p^0 = \{(W_1^0, f_1^0), \ldots, (W_n^0, f_n^0)\}$
2 $x \leftarrow 0$
3 **repeat**
4 **foreach** $i \in [1, n]$ **do**
5 $I_i^x \leftarrow DBAdv(W_i^x, f_i, b)$
6 Initialize $W_i^{x+1} = U$ for all $i \in [1, n]$
7 **foreach** $q \in Q$ **do**
8 Let $J_1^q, \ldots, J_n^q$ be a permutation of $I_1^l, \ldots, I_n^l$ such that $cost(q, J_1^q) \leq cost(q, J_2^q) \cdots \leq cost(q, J_n^q)$
9 **foreach** $j \in [1, m]$ **do** add q to W_i^{x+1} such that $I_i^x = J_j^q$
10 $p^{x+1} \leftarrow \{(W_1^{x+1}, f_1^{x+1}), \ldots, (W_n^{x+1}, f_n^{x+1})\}$
11 $x \leftarrow x + 1$
12 **until** $|TotalCost(p^{x-1}, m) - TotalCost(p^x, m)| < \epsilon$ *or* $x > IterMax$
13 **return** p

Figure 3: Pseudocode for DIVGDESIGN

Figure 2(b). Note that query q_2 appears in W_1^0 and yet has the least execution cost with configuration I_2^0. This may happen because q_2 is more "similar" to q_3 in terms of beneficial configurations, e.g., both benefit from the same indexes or materialized views, and hence I_2^0, which is computed based on q_3, is more effective for q_2 compared to I_1^0. Based on the computed costs, the algorithm moves each query to the sub-partition corresponding to the least cost. The resulting partition $\{W_1^1, W_2^1\}$ is shown in Figure 2(c), where q_2 now appears in the same sub-partition with q_3. This new partition becomes the input of the next iteration, and the iterations continue until a convergence condition (which we discuss later) is satisfied.

Overall, DIVGDESIGN tries to create partitions that group queries that are highly "similar" with respect to the configuration elements (e.g., indexes or materialized views) that benefit them. This strategy ensures that each resulting configuration $I_i = DBAdv(W_i, f_i, b)$ is beneficial for all the queries in W_i. Equally importantly, this similarity is computed indirectly through the output of the black-box function $DBAdv()$, without the need to model the logic of the configuration advisor.

4.2 Algorithm Definition

Figure 3 shows the pseudocode of the DIVGDESIGN algorithm that we introduce in this paper. The algorithm receives as input the workload W, the number of replicas n, the space budget b, and the load balancing factor m, and outputs an m-balanced design p such that $TotalCost(p, m)$ is small.

The algorithm picks (line 1) an initial design $p^0 = \{(W_1^0, f_1^0), \ldots, (W_n^0, f_n^0)\}$ and then refines it in an iterative fashion (lines 3–12). Keep in mind that the weight functions $f_1^0, \ldots, f_n^0$ are fully specified through the sub-partitions $W_1^0, \ldots, W_n^0$ given that p is an m-balanced design (Definition 2.4). Hence, it suffices to pick a random partition $\{W_1^0, \ldots, W_n^0\}$ in order to initialize p^0. We use x as

the variable that tracks the current iteration number, and use $p^x = \{(W_1^x, f_1^x), \ldots, (W_n^x, f_n^x)\}$ for the design at the beginning of the current iteration, and $p^{x+1} = \{(W_1^{x+1}, f_1^{x+1}), \ldots, (W_n^{x+1}, f_n^{x+1})\}$ for the design at the end of the current iteration. (As always, the weight functions are determined based on the sub-partitions according to Definition 2.4.) Each iteration comprises two steps. In the first step, the algorithm computes the configuration I_i^x of replica i by invoking the configuration advisor on W_i^x, i.e., $I_i^x = DBAdv(W_i^x, f_i^x, b)$. In the second step, the algorithm computes the new design based on these configurations. Specifically, for each query $q \in Q$, DIVGDESIGN computes $cost(q, I_i^x)$ for all $i \in [1, n]$, and then identifies the m configurations $J_1^q, \ldots, J_m^q$ from $I_1^x, \ldots, I_n^x$ that yield the m least costs for q. Then, q is added to W_i^{x+1} if I_i is among $J_1^q, \ldots, J_m^q$. Hence, each query is assigned to the m sub-partitions that correspond to the m most efficient evaluation plans for the query.

How do we know that this iterative process converges to a good design? One of our key contributions is to show that each iteration of DIVGDESIGN cannot worsen the quality of p under certain assumptions.

THEOREM 4.1. *Let p^x be the design at the beginning of iteration x (line 3 in Figure 3) and p^{x+1} be the design at the end of the iteration (line 10 in Figure 3). Assuming that $DBAdv()$ returns optimal configurations, it holds that $TotalCost(p^x, m) \geq TotalCost(p^{x+1}, m)$, for all $x \geq 0$.*

This theoretical result is interesting, since it relies solely on the optimality of $DBAdv()$ and does not make further assumptions about this black-box function. In practical terms, where $DBAdv()$ is imperfect but still identifies effective configurations, we have observed that each iteration yields a design p^{x+1} whose performance is either significantly better or close to p^x. (The details appear in Section 5.) Overall, the theoretical and empirical evidence suggest that the algorithm converges to a good design.

The convergence condition (line 12) checks that the $TotalCost()$ value of the new design does not differ from the previous design beyond a threshold ϵ, which is a parameter of the algorithm. Intuitively, this is a point where the algorithm has discovered a good divergent design, and subsequent iterations refine this design with diminishing returns. However, certain input instances may cause the algorithm to terminate only after an exponential number of iterations. To avoid such problematic cases, the convergence condition also imposes an upper bound on the number of iterations, based on a second parameter $IterMax$. Clearly, the two parameters control the trade-off between total execution time of the algorithm and performance of the output partition p. Our experimental study indicates $\epsilon = 1.0\%$ and $IterMax = 10$ yields a good balance point in practice.

To improve the chances of discovering a good divergent design, we invoke the algorithm several times, each time starting with a different initial design. In this fashion we collect several designs and return the one with the minimum $TotalCost()$ value.

An important detail of DIVGDESIGN is the choice of the initial partition p^0. In the current definition p^0 is set as a random partition (for each restart). There may be other methods of higher overhead and perhaps of higher potential to improve the final output which are worth investigating as future work. Another important point is that the design p^{x+1} may have sub-partitions that do not have any queries. This can happen in the generation of $W_1^{x+1}, \ldots, W_n^{x+1}$ at the end of the iteration, if no query in a previous part W_i^x has I_i^x among its best m configurations (line 9). Such designs are not desirable, as they reduce the effective number of replicas for query processing. Our current implementation of DIVGDESIGN simply aborts the current iteration if this happens and restarts the algorithm with a different random design. Our experiments (Section 5) show that random initialization and restarts work well in practice.

Parameter	Values
n	2, 3, 4
m	1, 2
Space budgets	0.25×, 0.5×, 1.0×, 2.0×, INF (infinite)

Table 2: Summary of experimental parameters.

5. EXPERIMENTAL STUDY

This section presents the results of an experimental study that we conducted in order to evaluate the effectiveness of the DIVGDESIGN algorithm and the performance of the divergent designs that it computes. We first discuss the methodology we followed for the experiments and then present our findings.

5.1 Methodology

Data Sets and Workloads We employ data sets and workloads from the standard benchmark TPC-H [16] and its successor TPC-DS [15]. These benchmarks comprise analytic workloads consisting of 22 and 99 query templates respectively. As a newer benchmark, TPC-DS includes more diverse and complex queries compared to TPC-H.

For both benchmarks, we create 10GB databases and corresponding query workloads using the supplied (data and query) generators with the default parameters. Each workload is created as a single query stream with all frequencies set to one, thus assigning equal weight to each template. For TPC-H, we create a workload with updates (termed TPC-H-u) using the 22 queries with *refresh* streams RF1 and RF2 at the rate of 0.1% as prescribed by the benchmark. These streams contain inserts and deletes respectively that are applied to both the LINEITEM and ORDERS tables. We create an additional update workload (termed TPC-H-u110) that has an increased proportion of queries to updates. Specifically, we concatenate five streams of 22 queries each, in addition to the refresh streams RF1 and RF2 mentioned above. We do not consider update workloads for TPC-DS, as their implementation according to the benchmark is quite complicated.

During our evaluation, we observed that TPC-DS queries 4 and 11 were so expensive that they effectively obscured all other queries in the workload. It appeared that the the DB2 Design Advisor was unable to recommend good indexes to improve those two queries, therefore their costs would remain the same in a uniform or divergent design. Moreover, each query of the two had a cost that exceeded the combined costs of the remaining 97 queries. For these reasons we exclude those two queries from our TPC-DS workload.

Experimental Parameters. Our experiments vary the following parameters: number of replicas n, load balancing factor m, and index configuration space budget b. Table 2 shows the values that we use in the experiments. The storage space budget is measured as a multiple of the base data size, i.e., given our 10 GB base data size, a space budget of 1.0× indicates a 10 GB storage space budget.

DIVGDESIGN Implementation. We completed a prototype implementation of DIVGDESIGN over IBM DB2 (Express-C version 9.7 for 64-bit Linux). As presented in Section 4.2, DIVGDESIGN utilizes the system's advisor (IBM DB2 Design Advisor, in this case) to compute per-replica configurations, and the what-if optimizer to estimate query costs under these configurations. In our implementation, we invoked the DB2 Design Advisor to compute index-only configurations. We note that our methods are applicable for materialized views as well. Since DIVGDESIGN treats the $DBAdv$ and

what-if query optimizer as black boxes, our implementation can be easily ported to another database system with these features.

We set the convergence parameters of DIVGDESIGN to $\epsilon = 1\%$ and $IterMax = 10$. We actually performed minimal tuning for these settings, and therefore they are not particularly tailored to the workloads that we employ. Given input parameters W, f, b, n, and m, we run the DIVGDESIGN algorithm five times and output the lowest cost design out of the all the independent runs (see also Section 4.2). We denote this final design as p_{divg}.

Metrics. We measure the performance of the divergent design p_{divg} (computed by DIVGDESIGN) primarily through the value of the total cost metric $TotalCost(p_{divg}, m)$. We compute this metric by invoking the what-if optimizer to estimate the costs of queries and updates under the configurations implied by p_{divg}. This methodology, which is consistent with previous studies on physical design tuning, allows us to gauge the effectiveness of DIVGDESIGN in isolation from any estimation errors in the optimizer's cost models. In several cases, we report the improvement of $TotalCost(p_{divg}, m)$ compared to $TotalCost(p_{unif}, m)$, which measures the performance of the uniform design (the current practice in real-world systems).

We also perform several experiments where we measure the wall-clock time to execute workloads using the configurations corresponding to p_{divg} (again, with IBM DB2 as the DBMS). In these experiments, we actually build the indexes corresponding to each replica. To ensure statistical robustness, we repeat these experiments three times and report the average execution time.

Experimental Platforms. In all experiments, DIVGDESIGN and IBM DB2 run on Ubuntu Linux 10.04 *LTS* inside a VMWare virtual machine. The virtual machine infrastructure provides the convenience of setting up identical environments over different host machines.

We employ two experimental platforms for the host machines. All experiments that measure the $TotalCost(p_{divg}, m)$ metric run in a single host machine with a dual-core Intel 2.6GHz CPU and 8GB of memory. All experiments that measure wall-clock time run on a local cluster of host machines, each having two dual-core AMD 2.0GHz CPUs and 8GB of RAM. The host machines in the local cluster are connected with a gigabit network and switch. In both platforms, the virtual machine running IBM DB2 is assigned 2 cores and 4GB of RAM.

5.2 Behavior of DIVGDESIGN

The first set of experiments examines the behavior of DIVGDESIGN and the characteristics of the divergent designs that it computes.

Iterative improvement in DIVGDESIGN. We first examine how each iteration of DIVGDESIGN improves the currently computed design. Theorem 4.1 states that each iteration cannot lead to a worse design if the DBMS advisor is optimal, so our goal is to examine the validity of this result under the imperfect DB2 Design Advisor. For this experiment, we employ the TPC-DS workload with $b = 0.25$, $n = 4$ and $m = 1$, but we obtained similar results for other values.

Figure 4 shows the $TotalCost(p, m)$ metric at each iteration of the algorithm, where p is the design at the end of the iteration. Each of the five curves represents an independent run of DIVGDESIGN with a different initial (random) partition. (Recall that p_{divg} is the best design out of these five runs, which is run 4 in this example.)

The results show that most iterations successfully improve the currently computed design, which matches the theoretical result of Theorem 4.1 even though the advisor is imperfect. In fact, run 4 shows a slight upward trend at the fourth iteration, which is due precisely to this imperfection. Still, DIVGDESIGN was able to recover from this setback in subsequent iterations, and this run resulted in the best design overall.

In all the experiments that we present, DIVGDESIGN converged to a solution within six iterations, well before the $IterMax$ cutoff of ten iterations. This also had a beneficial effect on running time. As an example, for the TPC-DS workload of the previous experiment (97 queries), DIVGDESIGN required around three minutes per iteration, leading to a maximum of 18 minutes per run. We observed similar run times for all other experiments.

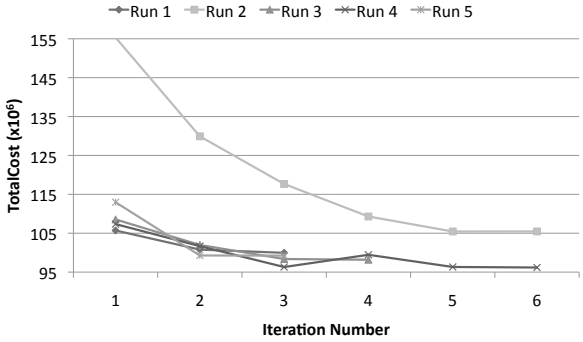

Figure 4: Five independent runs of DIVGDESIGN.

Diversity of output design p_{divg}. Next, we evaluate the diversity of the divergent design p_{divg} computed by DIVGDESIGN. Diversity is a primary differentiator for divergent designs and one of the key motivations behind their usage. Since our prototype implementation invokes the DB2 Design Advisor to compute index-only configurations, we measure the diversity of p_{divg} in terms of the number of distinct indexes contained in the corresponding configurations $I_1, \ldots, I_n$. As a yardstick for comparison, we also measure the number of distinct indexes in the configuration I_{unif} of the uniform design p_{unif} for the same number of replicas. Clearly, our expectation is that p_{divg} should contain a significant number of additional indexes compared to p_{unif}.

Table 3 reports the total count of distinct indexes for p_{divg} and p_{unif}, for input instances with $m = 1$. We select $m = 1$ because it corresponds to maximum specialization and hence maximal diversity. As shown, the number of additional indexes varies across the experiments, but overall the results demonstrate that p_{divg} contains significantly more indexes compared to p_{unif}. The value of these additional indexes will become evident in the experiments that follow. For a few cases we examined the indexes in p_{unif} that do not appear in p_{divg} and found that they were mostly indexes on smaller tables. Somewhat surprisingly, p_{divg} has a higher diversity even for the infinite space budget, but this is due to the imperfection of the DB2 Design Advisor.

5.3 Performance for Query-Only Workloads

The next set of experiments evaluates the performance of p_{dvg} on query-only workloads based on the $TotalCost()$ metric. Results with mixed workloads of queries and updates appear in the following sub-section.

Figure 5 shows the performance of p_{divg} as we vary the space budget and the number of replicas, for $m = 1$ and the TPC-H workload. Figure 6 shows the same results for TPC-DS. In both cases, we report the improvement of $TotalCost(p_{divg}, m)$ over $TotalCost(p_{unif}, m)$, where p_{unif} is again the baseline uniform design for the same number of replicas. The results demonstrate a wide range of gains with several interesting trends. The largest performance improvements for TPC-H are 16% and 20% for $n = 2$

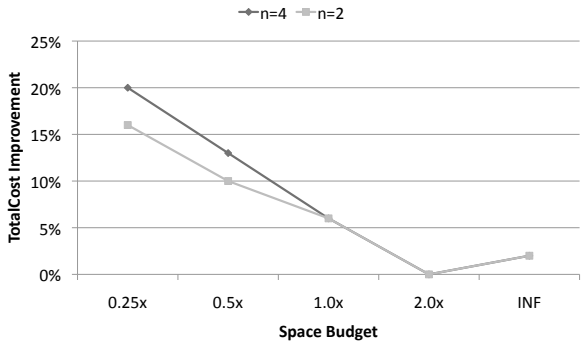

Figure 5: Performance of divergent design for TPC-H query-only workload and $m = 1$.

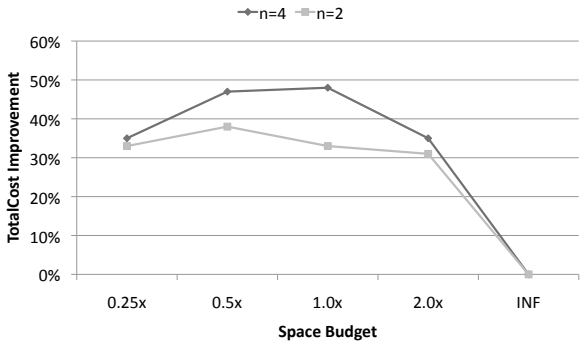

Figure 6: Performance of divergent design for TPC-DS query-only workload and $m = 1$.

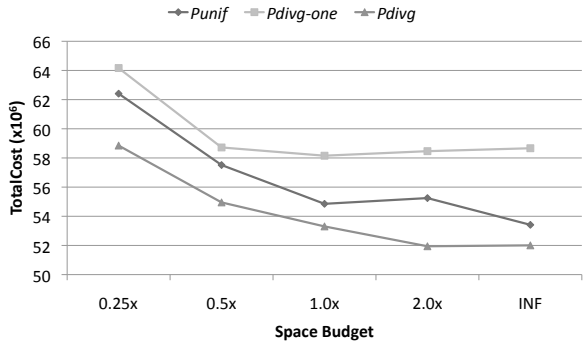

Figure 8: Performance of divergent designs for TPC-H query-only workload, $n = 4$, $m = 2$.

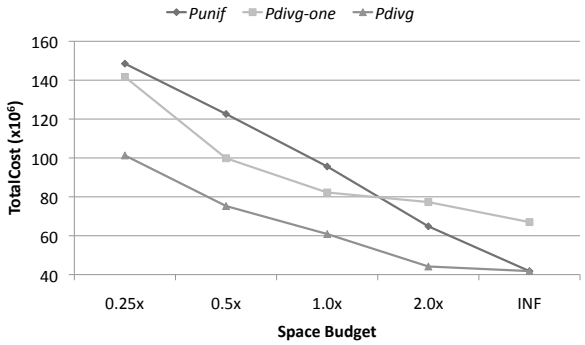

Figure 9: Performance of divergent designs for TPC-DS query-only workload, $n = 4$, $m = 2$.

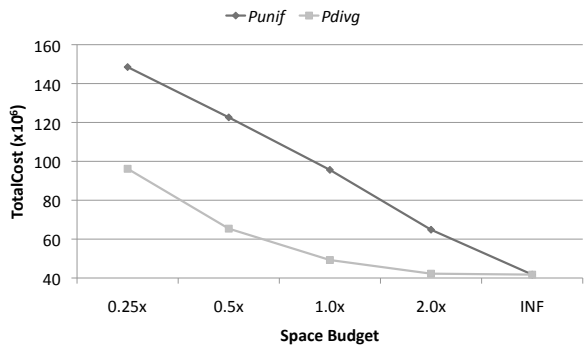

Figure 7: Performance of divergent design and uniform design for TPC-DS query-only workload, $m = 1$, $n = 4$.

and $n = 4$ respectively at the $0.25\times$ space budget. TPC-DS shows a 38% improvement for $n = 2$ at $0.5\times$ and 48% improvement for $n = 4$ at $1.0\times$ space budgets.

The general trend is that improvements increase at lower space budgets and also with a higher number of replicas. A divergent design makes better usage of the aggregate disk space for configurations, and can thus install more indexes than p_{unif} even at low space budgets (see also Table 3). Accordingly, as n increases, a divergent design can specialize each replica to a smaller subset of the workload, thus yielding better performance. It is interesting to note that the gains are consistently higher for TPC-DS than for TPC-H. TPC-DS is much more diverse and with far more beneficial indexes (see again Table 3), thus the divergent design is able to partition the workload and distribute the indexes more effectively.

The performance gains become smaller as the space budget grows, because at some point there is enough space to materialize all the beneficial indexes. We observed this to be near $2.0\times$ for TPC-H and $7.5\times$ for TPC-DS. An exception to this observation is the slight gain for the INF budget in Figure 5, which however can be attributed to the imperfect heuristics employed by the advisor. Overall, the improvements shown at each space budget appear to match well with the proportion of additional distinct index counts shown in Table 3.

	Space Budget				
	$0.25\times$	$0.5\times$	$1.0\times$	$2.0\times$	INF
TPC-H, $n = 2$					
p_{divg}	27	31	39	42	42
p_{unif}	22	23	35	40	42
TPC-H, $n = 4$					
p_{divg}	35	39	42	43	43
p_{unif}	22	23	35	40	42
TPC-DS, $n = 2$					
p_{divg}	138	165	191	229	275
p_{unif}	119	134	154	185	269
TPC-DS, $n = 4$					
p_{divg}	172	197	237	275	279
p_{unif}	119	132	152	184	268

Table 3: Number of distinct indexes in the configurations of p_{unif} and p_{divg}.

Figure 7 shows in detail the performance of p_{divg} and p_{unif} for the TPC-DS workload and $n = 4$. Here we chart the $TotalCost()$ metric of each design. We observe that the cost of the divergent design decreases rather smoothly with the larger space budgets as expected, exhibiting the same stable behavior as the uniform design. More importantly, we observe the following interesting pattern: the divergent design with a space budget of $0.25\times$ performs

nearly the same as the uniform design with a 1.0× space budget, and the same holds for the 0.5× divergent design versus the 2.0× uniform design, and for the 1.0× divergent design versus the 5.0× uniform design. In other words, given four replicas, the divergent design for TPC-DS can perform as well as the uniform design with only $\frac{1}{4}$ of the space budget per replica. (We obtained similar results with TPC-H.) This intuitive property validates the effectiveness of DIVGDESIGN in computing a good divergent design, even though the algorithm does not have any visibility in the structure of the workload (which is quite complicated in this experiment) or in the inner workings of the DBMS configuration advisor. Moreover, these results indicate that a divergent design may be particularly beneficial if the configuration space budget is restricted.

Our last set of experiments evaluates the effect of parameter m on the performance of the divergent designs computed by DIVGDESIGN. To illustrate the trade-off between performance and load-balancing, we consider one additional design in these experiments that we denote as p^{one}_{divg} and which is the design computed by DIVGDESIGN for $m = 1$. Clearly, p^{one}_{divg} has higher specialization than p_{divg} and can thus yield better performance if every query q can be routed to its best replica. On other hand, q may have a much higher cost on its 2nd-best replica in p^{one}_{divg}, which implies less flexibility in terms of load balancing.

Figure 8 shows the $TotalCost$ metric for designs p_{divg}, p_{unif} and p^{one}_{divg} for the TPC-H workload, $n = 4$, $m = 2$. Figure 9 shows the same results for TPC-DS. Overall, the results confirm our intuition: p^{one}_{divg} has worse performance than p_{divg}, and in most cases it performs even worse that the uniform design p_{unif}. These results indicate that the specialization in p^{one}_{divg} causes great imbalance for the cost of the same query across different replicas, thus leading to a high $TotalCost()$ metric for $m = 2$. On the other hand, p_{divg} creates a specialization of replicas that directly takes into account factor m, which in turn yields consistent improvements over the uniform design. The gains are small for TPC-H but very significant for TPC-DS, ranging from 32% to 39% over p_{unif}.

To summarize, our results demonstrate that the divergent designs computed by DIVGDESIGN outperform the baseline uniform design, often by a significant margin. The improvement is more pronounced with the TPC-DS workload, which is somewhat expected given that this workload is more complex than TPC-H and also because IBM DB2 is heavily optimized for the execution of TPC-H queries (as are many commercial systems).

5.4 Results with Mixed Workloads

Next we evaluate the performance of divergent designs with mixed workloads of queries and updates. In this case, any design has to balance the benefits of the per-replica configurations against the maintenance cost due to updates. The experiments that follow employ the TPC-H-u and TPC-H-u110 workloads, to evaluate the effect of the query-to-update ratio on the performance of divergent designs. We first present experiments with $m = 1$ (highest specialization) and then consider the effect of $m > 1$.

Figure 10 shows the percent improvement of p_{divg} over p_{unif}, for the two TPC-H workloads and $m = 1$. The gains are small for TPC-H-u, since the cost of updates leads the DB2 Design Advisor to recommend relatively few indexes for both designs. Conversely, the gains increase with the TPC-H-u110 workload, ranging from 19% to 33% for $n = 4$, since the higher percentage of queries justifies the creation of more indexes in the divergent design. We also observe an increase in gains as n grows, which is in accordance to the trend for query-only workloads.

To gain further insights on the gains of the divergent designs, we break down the improvement per queries and updates for TPC-H-u110 and $n = 4$. The results are shown in Figure 11. We observe that query performance is improved by 18% at $b = 0.25$, but from that point onwards queries have essentially the same performance between the divergent and uniform design. The big difference is in the cost of updates, where the improvement ranges from 30% to over 70%. The reason can be traced again to the asymmetry between queries and updates–a query can be executed by any single replica, whereas an update has to be executed by all replicas. Intuitively, the cost to apply the update is multiplied by n. The uniform design ignores this magnified effect and results in indexes whose maintenance cost is much higher than their benefits. In contrast, the divergent design p_{divg} takes this asymmetry directly into account, and results in indexes that strike a better balance between benefit and maintenance cost.

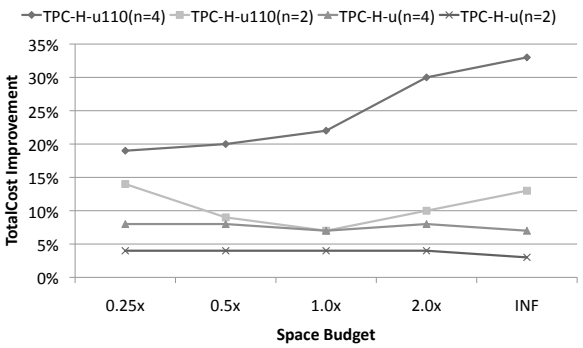

Figure 10: Performance of divergent designs for TPC-H with updates and $m = 1$.

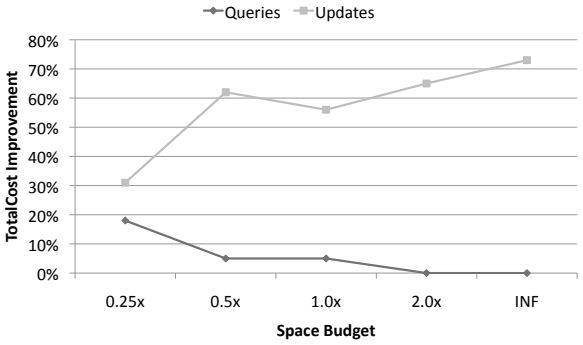

Figure 11: Percent improvement breakdown for queries and updates in TPC-H-u110, $n = 4$, $m = 1$.

Table 4 shows the number of indexes in p_{divg} and p_{unif} for $b = 1.0$ in the previous experiment, for the ORDERS and LINEITEM tables that receive all the updates. The reported counts support our previous observation. The divergent design places far fewer indexes across the replicas, which in turn leads to reduced maintenance cost (over 50% reduction, as shown in Figure 11). Moreover, we verified that p_{divg} comprises *all* the indexes of p_{unif}, except that they are distributed across different replicas. It is interesting to note that we observed similar results when we examined p_{divg} and p_{unif} for $b = INF$: the divergent design installed a total of 20 indexes, compared to 72 indexes for the uniform design. As shown in Figure 11, this allocation results in 70% improvement in index maintenance cost while retaining the same benefits for query evaluation. Thus, p_{divg} results in a more judicious allocation of indexes even when the per-replica configurations are not constrained by the space budget.

Additional experiments considered the effect of m on the performance of the divergent design computed by DIVGDESIGN. We ex-

Design	Table Name	#Indexes per Replica			
p_{unif}	ORDERS	5	5	5	5
	LINEITEM	7	7	7	7
p_{divg}	ORDERS	2	2	1	1
	LINEITEM	5	2	2	4

Table 4: Index counts of the ORDERS and LINEITEM tables per replica for p_{unif} and p_{divg} for TPC-H-updates-110, $n = 4$, $m = 1$, $b = 1.0$.

perimented with $m = 1$ and $m = 2$, using the same methodology as for query-only workloads. The results remained qualitatively the same and are hence omitted in the interest of space.

Overall, our results demonstrate the divergent designs computed by DIVGDESIGN continue to outperform the baseline uniform design. The improvement becomes more significant when the ratio of queries increases, and most of the gains come from a judicious allocation of indexes to replicas that reduces substantially the total maintenance cost.

5.5 Performance with Concurrent Execution

We conclude with a set of experiments that evaluate the performance of divergent designs in terms of query throughput. We set up a replicated TPC-H database using $n = 3$ replicas and IBM DB2 as the DBMS. We create several concurrent client processes, each executing a random permutation of a subset of the TPC-H workload (to be defined shortly). We tune the system according to a specific design, and then measure performance in terms of the total wall-clock time to evaluate the query streams of the concurrent clients. Hence, the execution time of the slowest client determines the overall performance.

We invoke DIVGDESIGN to compute a divergent design p_{divg} with $b = 1.0$ and $m = 2$, so that there is some flexibility to load balance across the $n = 3$ replicas while maintaining some specialization. As in the previous experiments, we will compare p_{divg} against the uniform design p_{unif} for the same values of b and n.

We select the workload in order to favor the execution of queries under p_{unif}. Specifically, out of the 22 TPC-H queries, we select a subset of 17 queries that have similar[1] execution times under p_{unif}. This property allows us to use a simple query routing policy that is likely to work well for p_{unif}: When some client wishes to execute some query q, it routes q to the replica which currently has the least number of concurrently executing queries. It is not difficult to see that this policy is likely to evenly load each replica under p_{unif}, since each query has the same execution cost on all replicas and all queries have similar execution costs. We employ a slightly modified routing policy for p_{divg}: the client routes q to the replica which currently has the least number of concurrently executing queries and is among the best two replicas for q. The routing policy here follows the intuition of the divergent design (queries are routed to their specialized replicas), but it is not optimal for load balancing because the same query may have different execution costs across different replicas and queries do not necessarily have similar execution costs overall. Overall, we create a best-case scenario for the uniform design and a potentially problematic scenario for the divergent design.

Table 5 shows the total improvement of p_{divg} over p_{unif} in terms of wall-clock time as we vary the number of concurrent TPC-H clients. (We discuss the TPC-DS numbers later.) The divergent design consistently outperforms the uniform design, with an overall improvement of up to 20% in terms of query throughput. This

[1]Within one standard deviation of the median.

	15 clients	30 clients	45 clients
TPC-H	11.5%	20.1%	19.5%
TPC-DS	15.8%	23.3%	31.5%

Table 5: Total execution time improvement of divergent over uniform design for 17 TPC-H and 20 TPC-DS Queries, $n = 3, m = 2, b = 1$, for 15, 30, and 45 concurrent clients.

magnitude is non-trivial for several reasons. First, the divergent design employs a suboptimal query routing policy that does not take into account the actual load of each replica. Second, the predicted improvement according to $TotalCost()$ is only 3%, which means that the non-linear effects of concurrent query execution magnify the benefits of divergent designs. Third, $m = 2$ implies that each query must have low execution cost on at least two out of the $n = 3$ replicas, limiting the specialization a divergent design can achieve. Finally, IBM DB2 (as many commercial systems) is heavily optimized to execute these benchmark workloads. Is it very significant that we are able to improve performance by 20% by simply tuning the configuration of each replica.

Beside improved execution time, the divergent design brings clear benefits to the time required to tune the system. Specifically, the divergent design required 50% less time to build the corresponding indexes on all replicas, since fewer indexes are allocated per replica while maintaining similar query benefits. (This is analogous to the results we saw in Table 4.) Accordingly, the time to refresh the optimizer's statistics[2] after the indexes are built is reduced by 14% . All in all, if we count the end-to-end time to build the indexes, setup the optimizer's statistics, and execute the workload through the concurrent clients, the divergent design improves on the uniform by 67% for 15 concurrent clients, and by 52% for 45 concurrent clients.

We performed the same experiment with TPC-DS by first using a similar method to select a 20-query subset, and then executing the full experiment as described. The results are included in Table 5 and show a 15.8%, 23.3% and 31.5% improvement for 15, 30, and 45 clients respectively, indicating the same performance trends as the TPC-H experiment.

6. IMPLEMENTATION CONSIDERATIONS

The experimental results of the previous section show clearly the performance benefits of divergent design tuning. In what follows, we consider the implications of implementing divergent designs in a DBMS that employs replication. Our assumption is that the DBMS already has the infrastructure to employ the baseline uniform design p_{unif}.

An immediate observation is that the DBMS can readily replace p_{unif} with the fully-balanced design p_{FB}, i.e., the output of the DIVGDESIGN algorithm for $m = n$. As in p_{unif}, p_{FB} prescribes the same configuration for each replica and hence the system can handle query routing and load balancing in exactly the same way. However, our analysis shows that p_{FB} can never perform worse than p_{unif}, and hence there is an immediate benefit to switching to divergent designs.

For $m < n$, the divergent design may assign a different configuration to each replica and hence the query routing policy is affected. Specifically, the DBMS must route an incoming query q to the currently-best replica, taking into account the load in the system and also the cost terms $cost(q, I_i)$ for each replica $i \in [1, n]$. These terms are already known for a query $q \in W$. For an unseen query $q \notin W$, it is possible to perform what-if optimization or to estimate these cost terms by mapping q to a similar query q' that

[2]This corresponds to the RUNSTATS command in DB2.

has already been analyzed (e.g., using the similarity metric of [8] or the general method of [13]).

The use of divergent designs also affects how the system responds to changes in node availability or the intensity of the workload. For instance, the failure of node i causes the queries in W_i to be (temporarily) left with $m-1$ "good" replicas and hence fewer load-balancing choices (until a node brings I_i up again). Similarly, scaling out the system implies that n increases, which likely causes the current divergent design to become suboptimal. In general, examining the interplay between divergent designs and elasticity is an interesting direction for future work.

7. RELATED WORK

Physical configuration tuning There has been a long line of research studies on the problem of tuning the physical configuration of a single DBMS [4, 6, 10, 11, 18]. The proposed methods typically take a representative workload W as input, and after some analysis they recommend a physical configuration that optimizes the evaluation of the workload according to the optimizer's estimates. Some of the proposed techniques have also found their way in commercial advisor tools that come bundled with the DBMS. However, the proposed methods work for a centralized system and do not take replication into account. Extending them to the setting of a replicated database is not obvious. Still, we show it is possible to leverage existing tools to compute good divergent designs, provided that we can compute an effective partition of the workload.

Chaudhuri et al.[8] propose a method to cluster the queries of a workload into disjoint subsets for the purpose of workload compression. Unfortunately, the partitioning generated by [8] provides an unspecified (and likely large) number of disjoint subsets, and hence cannot be directly applied to solve the divergent index tuning problem where the number of clusters is equal to the (small) number of replicas. Additionally, the methods in [8] do not consider key factors or the $TotalCost$ metric, such as the space budget constraint or the load-balancing factor m.

Shinobi [17] is a system that utilizes workload information to partition the data and selectively index data within each partition. This results in less expensive index maintenance and reorganization costs, by creating and dropping indexes on subsets of the data (the workload-based partitions) as the access patterns change. However, Shinobi does not address replication of partitions or different index configurations on replicas of the same partition, which is the problem that we examine in our work. In fact, our techniques can be used to determine which indexes to install on each replica, and then Shinobi can be responsible for maintaining only the fragments of these indexes that are important for the current workload patterns.

Physical data organization on replicas. Previous works also considered the idea of diverging the physical organization of replicated data. The technique of Fractured Mirrors [12] builds a mirrored database that stores its base data in a different physical organizations on disk (specifically, in a row-based and a column-based organization). To take advantage of this storage model, the query processor is modified to run query execution plans that can work on both formats of the data. Similarly, Distorted Mirrors [14] presents logically but not physically identical mirror disks for replicated data. Neither of these explores different physical organization of indexes and materialized views for each mirror.

8. CONCLUSIONS

In this paper, we introduced the novel paradigm of divergent design tuning for database systems that employ replication. We showed that the tuning problem is computationally hard, and hence proceeded to design a heuristic tuning algorithm based on a theoretical and empirical analysis of the space of divergent designs. Experimental results validated the effectiveness of the algorithm and demonstrated the many benefits of divergent designs in practice.

An interesting direction for future work is removing our treatment of DBAdv() as a black box. By leveraging specific functionality or properties of DBAdv(), it may be possible to develop a more effective divergent design tuning algorithm.

Acknowledgments. This work was supported in part by NSF grant IIS-1018914, DOE grant DE-SC0005428 and an IBM Faculty Development Award.

9. REFERENCES

[1] Amazon relational database service (amazon rds) `aws.amazon.com/rds`.

[2] P.A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D.B. Lomet, R. Manne, L. Novik, and T. Talius. Adapting microsoft sql server for cloud computing. In *ICDE*, pages 1255 –1263, 2011.

[3] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *ICDE*, 2007.

[4] N. Bruno and S. Chaudhuri. Constrained physical design tuning. *PVLDB*, 1(1):4–15, 2008.

[5] S. Chaudhuri, M. Datar, and V. Narasayya. Index selection for databases: A hardness study and a principled heuristic solution. *IEEE TKDE*, 16(11):1313–1323, 2004.

[6] S. Chaudhuri and V. R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. pages 146–155, 1997.

[7] S. Chaudhuri and V. R. Narasayya. AutoAdmin 'What-if' Index Analysis Utility. In *SIGMOD*, pages 367–378, 1998.

[8] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek Narasayya. Compressing SQL workloads. In *SIGMOD*, pages 488–499, 2002.

[9] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zaït, and M. Ziauddin. Automatic SQL Tuning in Oracle 10g. pages 1098–1109, 2004.

[10] D. Dash, N. Polyzotis, and A. Ailamaki. Cophy: a scalable, portable, and interactive index advisor for large workloads. *PVLDB*, 4(5):362–372, 2011.

[11] H. Kimura, G. Huo, A. Rasin, S. Madden, and S. B. Zdonik. Coradd: Correlation aware database designer for materialized views and indexes. *PVLDB*, 3(1):1103–1113, 2010.

[12] R. Ramamurthy, D. J. DeWitt, and Q. Su. A case for fractured mirrors. *The VLDB Journal*, 12(2):89–101, 2003.

[13] P. Sarda and J. R. Haritsa. Green query optimization: Taming query optimization overheads through plan recycling. In *VLDB*, pages 1333–1336, 2004.

[14] J.A. Solworth and C.U. Orji. Distorted mirrors. In *Parallel and Distributed Information Systems*, pages 10–17. IEEE, 1991.

[15] Transaction Peformance Council. TPC-DS Benchmark.

[16] Transaction Performance Council. TPC-H Benchmark.

[17] E. Wu and S. Madden. Partitioning techniques for fine-grained indexing. In *ICDE*, pages 1127–1138, 2011.

[18] D. C. Zilio, J. Rao, S. Lightstone, G.M. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB*, pages 1087–1097, 2004.