# Locality-Aware Distribution Schemes

**Bruhathi Sundarmurthy**
University of Wisconsin-Madison, Madison, WI, USA

**Paraschos Koutris**
University of Wisconsin-Madison, Madison, WI, USA

**Jeffrey Naughton**
University of Wisconsin-Madison, Madison, WI, USA

**Abstract**

One of the bottlenecks in parallel query processing is the cost of shuffling data across nodes in a cluster. Ideally, given a distribution of the data across the nodes and a query, we want to execute the query by performing only local computation and no communication: in this case, the query is called parallel-correct with respect to the data distribution. Previous work studied this problem for Conjunctive Queries in the case where the distribution scheme is oblivious, i.e., the location of each tuple depends only on the tuple and is independent of the instance. In this work, we show that oblivious schemes have a fundamental theoretical limitation, and initiate the formal study of distribution schemes that are locality-aware. In particular, we focus on a class of distribution schemes called co-hash distribution schemes, which are widely used in parallel systems. In co-hash partitioning, some tables are initially hashed, and the remaining tables are co-located so that a join condition is always satisfied. Given a co-hash distribution scheme, we formally study the complexity of deciding various desirable properties, including obliviousness and redundancy. Then, for a given Conjunctive Query and co-hash scheme, we determine the computational complexity of deciding whether the query is parallel-correct. We also explore a stronger notion of correctness, called parallel disjoint correctness, which guarantees that the query result will be disjointly partitioned across nodes, i.e., there is no duplication of results.

**2012 ACM Subject Classification** Theory of computation → Database query processing and optimization (theory)

**Keywords and phrases** partitioning, parallel correctness, join queries

**Digital Object Identifier** 10.4230/LIPIcs.ICDT.2021.22

**Funding** *Paraschos Koutris*: NSF CRII-1850348.

## 1 Introduction

Modern data management systems utilize the power of parallelism to efficiently process huge datasets. These systems can often scale to hundreds, and even thousands of machines. However, as the data and scale increases, massively parallel systems face a critical bottleneck: the cost of communicating (or shuffling) data across different machines. The amount of data shuffling required to process a given query depends on the initial data distribution, or data partitioning. Given a query, it is desirable to obtain the query result with the minimum possible shuffling of data, while making sure that no machine is overloaded. Ideally, we can execute the query without any shuffling, by simply running it on the local fragment of each machine/node, and then taking the union of all the results. This notion of being able to execute queries with no data shuffling is called *parallel correctness*, or p-correctness for short, and was first introduced in [3].

Previous works [3, 4, 9, 12, 13, 18] have studied the problem of p-correctness for different classes of queries, including Conjunctive Queries (CQs), Unions of Conjunctive Queries (UCQs), and Datalog programs. In all prior work, the data distribution is specified by a *distribution policy*, where the location of each tuple depends only on the tuple, and not on the rest of the database. A distribution policy can hence be captured by a function $\mathbf{P}(t)$ that maps a tuple $t$ to a set of machines/nodes. Typical examples of distribution policies are hash partitioning, where each relation in the database is hashed on a chosen subset of its attributes, as well as hypercube partitioning [2, 3, 8, 12]. The latter distribution policy is used for load-optimal single round algorithms that compute CQs [5, 15], when the initial data distribution is arbitrary and thus any correct algorithm must take into account that there is no knowledge of where the data lies.

In this paper, we consider a more general class of distribution schemes, captured by a function $\mathbf{P}(t, I)$ that allocates a tuple using (possibly) additional information from the instance. Distribution schemes that ignore $I$ are called *oblivious*, and correspond to distribution policies. Oblivious distribution schemes, although simple to understand, can often lead to suboptimal partitioning, since the locality of the data across relations (or within the same relation) is not exploited at all. As we will show in Section 3, any oblivious scheme that guarantees p-correctness for some query $q$ incurs a certain storage overhead that is unavoidable regardless of the underlying instance.

To overcome this barrier, it is necessary to look into distribution schemes that are *locality-aware*, in the sense that tuples are distributed taking into account tuples in other relations. In practice, parallel data management systems partition data in smart ways [16, 17, 21] and several parallel large-scale systems are deploying locality-aware data partitionings [7, 19, 20, 22, 23] to minimize, or even reduce to zero, the amount of data shuffling for a given query or query workload.

Since it is generally infeasible to describe efficiently a non-oblivious distribution scheme, in practice we are interested in schemes that can be concisely represented. In this work, we will focus on a particular class of locality-aware distribution schemes that has been widely adopted in practice called *co-hashing*. In such a scheme, a set of relations is initially hash-partitioned. The tuples from the remaining relations are then collocated with tuples from other relations according to specified join conditions. We illustrate co-hashing with an example below:

▶ **Example 1.** Consider two binary relations $R(A_1, A_2)$ and $S(B_1, B_2)$. A co-hash scheme partitions $R$ by hashing on attribute $A_1$, and then distributes each tuple $s$ from $S$ to all nodes that consist of a tuple $r$ from $R$ that joins with $s$ on the join condition $R.A_2 = S.B_1$. The tuples of $S$ that do not join with any tuple in $R$ are hashed on attribute $B_1$. The scheme is locality-aware since the nodes where a tuple from $S$ is assigned depend on the tuples that occur in $R$. It is easy to observe that, given the above co-hash scheme, the query $q(x, y, z) = R(z, x), S(x, y)$ is parallel-correct.

In a general co-hash scheme, we can chain together arbitrarily many relations, as long as we are not introducing a cyclic dependency. Note also that a co-hash scheme strictly generalizes (oblivious) schemes where each relation is hashed independently. Though co-hashing can make join processing very efficient, determining whether a query is parallel-correct for a given co-hash distribution scheme can be challenging, as illustrated in the example below:

▶ **Example 2.** Consider the setup from Example 1, together with a third table $U(D_1)$ that is co-hashed with $R$ on $R.A_2 = U.D_1$. Let the tuples of $U$ that do not join with $R$ be hashed on attribute $D_1$. Now, consider the Conjunctive Query $q(x, y) = S(x, y), U(x)$. As we will see later, this query is parallel correct for this co-hash scheme for any input instance.

Consider the naive approach of checking p-correctness for a CQ: look at one binary join at-a-time, and check if this join appears somewhere in the co-hash scheme. This approach is employed by the current state of the art technique [23], but it fails in this example, since

$S, U$ are not directly connected in the co-hash scheme (but only through $R$). Hence, the system will conclude that data shuffling must be performed, even if it is not necessary. We should note that p-correctness depends critically on how the non-joining tuples are handled. For instance, if we choose to hash the non-joining tuples from $S$ on attribute $B_2$ instead, $q$ is not p-correct any more.

The notion of p-correctness guarantees that the union of the local results reconstructs the query result. However, a query result may end up in multiple nodes, so obtaining the final result may require an additional deduplication step. If we want to avoid this, we need to check for a stronger condition, *parallel disjoint correctness*, or pd-correctness, which determines whether tuples of a query result are disjointly partitioned across the nodes.

In addition to the correctness notions, we introduce and study in the context of co-hash schemes two additional properties. The first property, *obliviousness*, tests whether a particular relation can be oblivious in an otherwise non-oblivious scheme. This is important, because such a relation can be easily maintained in a parallel setting. The second property, *clustering*, tests whether tuples of a given relation that agree on a set of attributes are always located in the same unique node. For example, relation $R$ from Example 1 is clustered w.r.t. attribute $A_1$. Note that this implies that any query that groups by $A_1$ and aggregates can be computed locally without any data shuffling. Clustering with respect to all attributes in a relation $R$ is equivalent to *non-redundancy* (i.e., each tuple in $R$ is assigned to a unique node). For instance, relation $S$ from Example 1 exhibits redundancy, since an $S$-tuple can end up in two (or more) nodes.

**Our Contributions.** In this paper, we study the formal foundations for locality-aware distribution schemes. We next summarize our contributions.

1. **Theoretical Framework.** We introduce a general framework that captures locality-aware distribution schemes. We describe several desirable properties, including p-correctness, pd-correctness, obliviousness and clustering. Within this framework, we show that oblivious schemes have a fundamental barrier on how well they can localize data.
2. **Co-hash Schemes.** We formalize the class of co-hash distribution schemes, which is widely used in practice, by introducing the notion of a *co-hash graph* to concisely capture how the input instance is distributed.
3. **Deciding Properties**: We study three properties of co-hash schemes: balancedness, obliviousness and clustering. We first show how co-hash schemes can overcome the limitations of oblivious schemes. Then, we show that we can decide both obliviousness and clustering in polynomial time (in the size of the co-hash graph). We observe that taking functional dependencies into account can lead to better reasoning about whether a relation is clustered or not.
4. **Parallel correctness**: We study the complexity of determining parallel correctness for co-hash schemes for the class of CQs. We distinguish two subproblems, depending on whether we consider a specific instance, or we want to determine p-correctness across all possible instances. We show that the former subproblem is $\Pi_P^2$-complete for CQs, but coNP-complete when restricted to full CQs (CQs without projections). For the latter subproblem, we show that it is NP-hard for general CQs, while for full CQs the complexity drops to polynomial time.
5. **Parallel disjoint correctness**: Finally, we provide results for the complexity of pd-correctness for CQs. Results for the instance-specific subproblem follows from p-correctness. For the instance-independent subproblem, we show that pd-correctness for full CQs can be determined in polynomial time (while it remains NP-hard for general CQs).

## 2 Preliminaries

**Basics.** We adopt the named definition of a relation: a relation is of the form $R(A_1, \dots, A_r)$. Here, $R$ is the relation name, and $A_1, \dots, A_r$ are the attributes of the relation; we assume that the attributes are disjoint across different relations. We say that $\mathtt{ar}(R) = r$ is the *arity* of the relation. We denote the attribute set of a relation $R$ by $\mathsf{att}(R)$. We also associate with each relation $R$ a collection of *functional dependencies*, which we denote by $\mathsf{fd}(R)$. Given a subset of attributes $\mathbf{A} \subseteq \mathsf{att}(R)$, we denote by $\mathbf{A}^+$ the *fd closure* of $\mathbf{A}$ w.r.t. the functional dependencies in $\mathsf{fd}(R)$. A *database schema* $\Sigma$ is a finite collection of relations.

We assume a (possibly infinite) domain **dom**. An *instance* of a relation $R$ is a finite set of tuples of the form $R(a_1, \dots, a_r)$, where $a_i \in \mathbf{dom}$, and $r$ is the arity of relation $R$. Given a tuple $t = R(a_1, \dots, a_r)$, and an attribute $A_i$, we write $t[A_i]$ to denote the value of $t$ at the position $A_i$, i.e. $t[A_i] = a_i$. We naturally extend this notation to $t[\mathbf{A}]$, where $\mathbf{A} \subseteq \mathsf{att}(R)$. A *database instance* $I$ over a schema $\Sigma$ is a collection of relation instances $R^I$ for each relation $R$ in the schema $\Sigma$.

**Join Condition.** Given two relations $R, S$, we define a *join condition* $\lambda$ between $R, S$ to be a symmetric binary relation over $\mathsf{att}(R) \cup \mathsf{att}(S)$. Whenever $(A, B), (B, A) \in \lambda$, we will simply write $A = B$. A join condition corresponds to an equi-join between $R, S$: for example, the join condition $\{A_1 = B_1, A_2 = B_2\}$ describes the equi-join $R \bowtie_{A_1=B_1 \wedge A_2=B_2} S$. This formalization allows a join condition to contain equality on predicates that belong in the same relation. Given a binary relation $\lambda$, we denote by $\lambda^{\oplus}$ the minimum equivalence relation that contains $\lambda$.

▶ **Example 3.** Consider two relations $R(A, B), S(C, D)$ and the join condition $\lambda = \{A = C, B = C\}$. Then, the equivalence relation $\lambda^{\oplus}$ is $\{A = A, B = B, C = C, A = C, B = C, A = B\}$.

**Conjunctive Queries.** A CQ is an expression of the form $q(\mathbf{y}) = R_1(\mathbf{x}_1), \dots, R_\ell(\mathbf{x}_\ell)$. The tuples $y$ and $\mathbf{x}_1, \dots, \mathbf{x}_\ell$ consist of variables and/or constants. Here, $q(\mathbf{y})$ is called the head, $R_1(\mathbf{x}_1), \dots, R_\ell(\mathbf{x}_\ell)$ are called atoms and form the body. The symbols $\mathbf{y}$, $\mathbf{x}_1 \dots \mathbf{x}_\ell$ are vectors that may contain variables or constants. The variables in the head must be a subset of the variables that appear in the body. A CQ is *full* if every variable in the body appears in the head as well, and it is *boolean* if the head contains no variables, i.e., it is of the form $Q()$. A valuation $v$ is a mapping from the variables in $q$ to the constants in **dom**. We extend $v$ to be the identity mapping for constants. A valuation $v$ satisfies $q$ on instance $I$ if for every $i = 1, \dots, \ell$, $v(\mathbf{x}_i) \in I$. We define the output $q(I)$ to be the set of all $v(\mathbf{y})$, for a valuation $v$ that satisfies $q$ on $I$.

Given a CQ $q$, a *fractional edge packing* is an assignment of a non-negative weight $w_i$ to each atom $R_i$ such that for each variable $x_i$ in $q$, the sum of the weights of the atoms that contain $x_i$ is at most 1. The fractional edge packing number $\tau^*(q)$ is the maximum value of the quantity $\sum_i w_i$ over all possible fractional edge packings of $q$.

## 3 Data Distribution Schemes

Let $I$ be a database instance over a schema $\Sigma$. Given a set of $p$ nodes, $\mathcal{N} = \{1, 2, \dots, p\}$, our goal is to distribute the tuples of $I$ over these $p$ nodes, such that each node $i \in \mathcal{N}$ receives a subset $I_i \subseteq I$.

▶ **Definition 4** (Distribution Scheme). *Let $\mathcal{N} = \{1, 2, \ldots, p\}$ be a set of nodes, and $\Sigma$ a schema. A* distribution scheme $\mathbf{P}$ *is a function that takes as input an instance $I$ over the schema $\Sigma$ and a tuple $t \in I$, and returns a set $\mathbf{P}(t, I) \subseteq \mathcal{N}$.*

Given a distribution scheme $\mathbf{P}$, we define the *data chunk* for node $i$ as $\mathbf{P}^{(i)}(I) = \{t \in I \mid i \in \mathbf{P}(t, I)\}$. A distribution scheme can assign a tuple to the empty set, or even replicate a tuple by assigning to multiple nodes. A distribution scheme is deterministic, but in practice we often want to introduce randomness in how the input is distributed. We model this by considering a *family of distribution schemes* $\mathcal{P} = \{\mathbf{P}_1, \mathbf{P}_2, \ldots\}$ defined over the same set of nodes $\mathcal{N}$ and the same schema $\Sigma$. Intuitively, to distribute the data we will choose a distribution scheme from the family $\mathcal{P}$ uniformly at random.

## 3.1 Properties of Distribution Schemes

We next introduce several important properties that a distribution scheme can satisfy.

**Obliviousness.** A desirable property of a distribution scheme is that the set of nodes where each tuple is assigned depends only on the relation it belongs to and its attribute values, and not the entire instance.

▶ **Definition 5** (Obliviousness). *Let $\Sigma$ be a schema, and $R \in \Sigma$. We say that $R$ is* oblivious *w.r.t. a family of distribution schemes $\mathcal{P}$ if for every $\mathbf{P} \in \mathcal{P}$ it holds that $\mathbf{P}(t, I) = \mathbf{P}(t, I')$ for every tuple $t$ and pair of instances $I, I'$ over schema $\Sigma$ such that $t \in R^I, R^{I'}$.*

If every relation of the schema is oblivious w.r.t. $\mathcal{P}$, we say that $\mathcal{P}$ is oblivious, and we can simply express the distribution function as $\mathbf{P}(t)$ (for every $\mathbf{P} \in \mathcal{P}$). Such a scheme is referred to as a *distribution policy* in [3, 4]. A standard example of an oblivious distribution scheme is *hash-partitioning*, where each relation in the schema is hash-partitioned (according to a subset of the attributes) independently of the other relations. Another example is the Hypercube distribution scheme discussed in [3, 15].

An advantage of a distribution policy is that the location of each tuple can be decided by just examining the particular tuple. On the other hand, as we will see later in this section, a distribution policy often limits the way we can distribute data among the nodes, especially when we want to increase the locality of the data in terms of join computation. Recent work [23, 22, 7] has introduced distribution schemes that are not oblivious, and are designed to support join computation locally.

**Clustering.** A second desirable property of a distribution scheme is that the tuples of a relation $R$ are *clustered* with respect to a set of attributes in $R$.

▶ **Definition 6** (Clustering). *Let $\Sigma$ be a schema, $R \in \Sigma$ and $\mathbf{A} \subseteq \mathsf{att}(R)$. We say that $R$ is* $\mathbf{A}$-clustered *w.r.t. a family of distribution schemes $\mathcal{P}$ over a set of nodes $\mathcal{N}$ if for every $\mathbf{P} \in \mathcal{P}$, for every instance $I$, and any two tuples $t, t' \in R^I$ such that $t[\mathbf{A}] = t'[\mathbf{A}]$, there exists a (unique) node $n \in \mathcal{N}$ such that $\mathbf{P}(t, I) = \mathbf{P}(t', I) = \{n\}$.*

In other words, if $R$ is $\mathbf{A}$-clustered, then all the tuples that have the same values for $\mathbf{A}$ (i.e., are in the same group) always end up in the same node (and only one). In practice, this means that any `group-by` query on $R$ where the grouping attributes are a subset of $\mathbf{A}$ can be computed locally, without any data shuffling. In the special case where $\mathbf{A} = \mathsf{att}(R)$, being $\mathbf{A}$-clustered simply means that for every tuple $t$, $|\mathbf{P}(t, I)| = 1$, or in other words that there is *no redundancy* in the distribution of the tuples in $R$. In this case, we will say that $R$ is *non-redundant* w.r.t. to $\mathcal{P}$.

**Balancedness.** A third desirable property of a distribution scheme is that the storage overhead is small, and the data is partitioned across the nodes in a balanced way. A first attempt to model this would be using the *replication factor*, defined as $r(\mathcal{P}, I) = \max_{\mathbf{P}\in\mathcal{P}} \sum_i |\mathbf{P}^{(i)}(I)|/|I|$. However, the trivial distribution scheme that sends all tuples to a single node has perfect locality and the smallest possible replication factor ($r = 1$). To overcome this problem, we define the notion of balancedness:

▶ **Definition 7** (Balancedness). *Let $\Sigma$ be a schema, and $\mathcal{N}$ be a set of $p$ nodes. The* balancedness *of a family of distribution schemes $\mathcal{P}$ over $\Sigma$ and $\mathcal{N}$, and an instance $I$ is*

$$b(\mathcal{P}, I) = \frac{p}{|I|} \cdot \mathbb{E}_{\mathbf{P}\sim\mathcal{P}} \left[\max_{i\in\mathcal{N}} |\mathbf{P}^{(i)}(I)|\right].$$

*Here, $\mathbf{P} \sim \mathcal{P}$ means we sample uniformly at random a distribution scheme from $\mathcal{P}$.*

Intuitively, the balancedness tells us how much larger the (expected) maximum-sized chunk is compared to $|I|/p$, which is what a perfect splitting of the data would achieve. Note that we can define the balancedness at a relation-level as well. The lemma below gives some intuition about the values that balancedness can take: it is at least as large as the replication factor, but cannot exceed $p$.

▶ **Lemma 8.** $r(\mathcal{P}, I) \leq b(\mathcal{P}, I) \leq p$

## 3.2 Queries over Distribution Schemes

Given a distribution of an instance over the nodes, an important question is how a query $q$ can be computed over the given distribution. In this section, we introduce two notions that capture when a distribution scheme is amenable to efficient distributed query computation.

First, we ask whether it is possible to compute a query $q$ by performing exclusively local computation, without any data shuffling. In the case that this is possible, we say that $q$ is *parallel correct*, following the definition in [3].

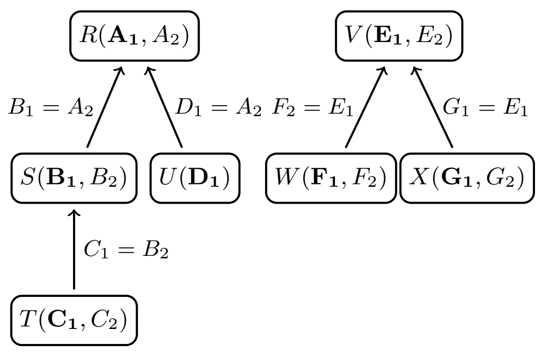
▶ **Definition 9** (Parallel Correctness). *Let $\mathcal{P}$ be a family of distribution schemes over schema $\Sigma$ and $\mathcal{N} = \{1, \ldots, p\}$ nodes. A query $q$ is* parallel correct (p-correct) *on instance $I$ w.r.t. $\mathcal{P}$ if for every $\mathbf{P} \in \mathcal{P}$, we have $q(I) = \bigcup_{i=1}^{p} q(\mathbf{P}^{(i)}(I))$. A query $q$ is p-correct w.r.t. $\mathcal{P}$ if it is p-correct w.r.t. $\mathcal{P}$ on every instance $I$ over $\Sigma$.*

Parallel correctness implies that the query result is the union of the local query results at each node. However, it is possible that a tuple is present in the local results of multiple nodes (i.e., there is redundancy in the result). In this case, getting the correct result would require a deduplication step after the result has been computed. To capture the case where deduplication is not necessary, we need to define a stronger notion of parallel-correctness.

▶ **Definition 10** (Parallel Disjoint Correctness). *Let $\mathcal{P}$ be a family of distribution schemes over schema $\Sigma$ and $\mathcal{N} = \{1, \ldots, p\}$ nodes. A query $q$ is* parallel disjoint correct (pd-correct) *w.r.t. $\mathcal{P}$ if for every $\mathbf{P} \in \mathcal{P}$ and for every instance $I$ over $\Sigma$, $\{q(\mathbf{P}^{(i)}(I))\}_{i\in\mathcal{N}}$ is a partition of the query result $q(I)$.*

## 3.3 Limitations of Oblivious Schemes

An ideal distribution scheme should be easy to compute (oblivious), cheap to store (balancedness), and also able to minimize the amount of data shuffling in order to compute a given query (parallel-correctness). With the next result, we show that there is a fundamental limitation on what an oblivious distribution scheme can achieve. This result is a straightforward corollary of existing lower bounds on parallel query evaluation [6].



**(a)** Co-hash graph.

R

| $A_1$ | $A_2$ |
|---|---|
| 1 | 2 |
| 1 | 3 |
| 2 | 3 |
| 3 | 4 |

S

| $B_1$ | $B_2$ |
|---|---|
| 2 | 4 |
| 2 | 5 |
| 3 | 5 |
| 8 | 9 |

T

| $C_1$ | $C_2$ |
|---|---|
| 4 | 8 |
| 5 | 7 |
| 5 | 8 |

U

| $D_1$ |
|---|
| 2 |
| 3 |
| 8 |

V

| $E_1$ | $E_2$ |
|---|---|
| 1 | 5 |
| 2 | 6 |
| 8 | 7 |

W

| $F_1$ | $F_2$ |
|---|---|
| 2 | 1 |
| 1 | 2 |
| 8 | 3 |

X

| $G_1$ | $G_2$ |
|---|---|
| 2 | 8 |
| 3 | 1 |
| 8 | 6 |

**(b)** Database instance.

R

| $A_1$ | $A_2$ |
|---|---|
| 1 | 2 |
| 1 | 3 |
| 2 | 3 |
| 3 | 4 |

S

| $B_1$ | $B_2$ |
|---|---|
| 2 | 4 |
| 2 | 5 |
| 3 | 5 |
| 3 | 5 |
| 8 | 9 |

T

| $C_1$ | $C_2$ |
|---|---|
| 4 | 8 |
| 5 | 7 |
| 5 | 7 |
| 5 | 8 |
| 5 | 8 |

U

| $D_1$ |
|---|
| 2 |
| 3 |
| 3 |
| 8 |

V

| $E_1$ | $E_2$ |
|---|---|
| 1 | 5 |
| 2 | 6 |
| 8 | 7 |

W

| $F_1$ | $F_2$ |
|---|---|
| 2 | 1 |
| 1 | 2 |
| 8 | 3 |

X

| $G_1$ | $G_2$ |
|---|---|
| 2 | 8 |
| 3 | 1 |
| 8 | 6 |

**(c)** Induced data distribution.

**Figure 1** Depiction of a co-hash graph and the resulting data distribution. Figure (c) shows the distribution of tuples induced by each of the co-hash trees. Each color corresponds to a different node.

▶ **Theorem 1.** *Let $\mathcal{P}$ be an oblivious family of distribution schemes over schema $\Sigma$ and nodes $\mathcal{N} = \{1, \dots, p\}$. If a self-join-free join query $q$ is parallel-correct w.r.t. $\mathcal{P}$, then for every instance $I$, we have $b(\mathcal{P}, I) = \Omega(p^{1-1/\tau^*(q)})$, where $\tau^*(q)$ is the fractional edge packing number of $q$.*

It is important to note that the bound holds for *any instance* $I$, even if the given instance is easy to distribute efficiently. We should also remark that Theorem 1 holds even for distribution schemes where a tuple $t \in R^I$ is distributed by taking all of $R^I$ into account (but not any other relation). Hence, to overcome the lower bound from Theorem 1, the distribution scheme has to use information from other relations in the instance.

▶ **Example 11.** Consider the conjunctive query $q(x, y, z, w) = R(x, y), S(y, z), T(z, w)$. The maximum fractional edge packing for $q$ is $\tau^* = 2$, hence Theorem 2 implies that any oblivious scheme such that $q$ is p-correct must have balancedness $\Omega(p^{1/2})$ for any instance $I$. We will see in the next section how locality-aware distribution schemes can overcome this bound.

## 4 Co-Hashing

We now introduce a class of non-oblivious distribution schemes, which we call *co-hash schemes*, that have been widely used in several practical settings [23, 22, 7].

### 4.1 Formal Framework

We start by recalling the definition of a rooted in-tree. A *rooted in-tree* is a directed tree such that $(i)$ a single designated vertex is called the *root*, and $(ii)$ every other vertex points *towards* the root.

▶ **Definition 12** (Co-Hash Graph). *Let $\Sigma$ be a schema. A* co-hash graph $\mathbb{G} = (V, E, \alpha, \lambda)$ *for $\Sigma$ is an edge-labelled and vertex-labelled directed graph where:*

- *the vertex set $V$ consists of relations in $\Sigma$;*
- *the label of a vertex $R$, denoted $\alpha(R)$, is a vector of attributes from* $\mathsf{att}(R)$*;*
- *the label of an edge $e = (R, S)$, denoted $\lambda(e)$, is a join condition between $R, S$; and*
- *the edge set $E$ consists of the union of the disjoint sets of edges of the rooted in-trees.*

Each rooted in-tree in $\mathbb{G}$ is called a *co-hash tree*. The *root path* of a vertex $R$ is the unique directed path from $R$ to the root of its co-hash tree. A co-hash graph does not need to include as vertices all relations from the schema. We will refer to the vertex label $\alpha(R)$ as the *hash signature* of $R$, and denote the attribute at the $i$-th position as $\alpha(R)_i$.

▶ **Example 13.** Consider the schema: $\Sigma = \{R(A_1, A_2), S(B_1, B_2), T(C_1, C_2), U(D_1)\}$, and a co-hash tree with four vertices $R, S, T, U$. Let the vertex labels be: $\alpha(R) = \langle A_1 \rangle$, $\alpha(S) = \langle B_1 \rangle$, $\alpha(T) = \langle C_1 \rangle$, $\alpha(U) = \langle D_1 \rangle$.

There are three edges: $e_{SR} = (S, R)$ with join condition $\lambda(e_{SR}) = \{B_1 = A_2\}$, $e_{TS} = (T, S)$, with join condition $\lambda(e_{TS}) = \{C_1 = B_2\}$, and $e_{UR} = (U, R)$ with join condition $\lambda(e_{UR}) = \{D_1 = A_1\}$. Figure 1a depicts the above co-hash tree.

**Induced Distribution.** We now explain how a co-hash graph $\mathbb{G}$ induces a family of distribution schemes $\mathcal{P}_{\mathbb{G}}$ over a set of nodes $\mathcal{N}$. Let $I$ be an instance of the schema $\Sigma$. For a tuple $t \in R$, we define its *upwards join set* $\mathcal{J}_{\uparrow}^{\mathbb{G}}(t, I)$ in $\mathbb{G}$ as follows. If $R$ is a root of some co-hash tree, then $\mathcal{J}_{\uparrow}^{\mathbb{G}}(t, I) = \emptyset$. If $R$ is not the root, then let $S$ be its unique parent in the co-hash tree. Then, $\mathcal{J}_{\uparrow}^{\mathbb{G}}(t, I)$ is the set of all tuples from $S$ that join with $t$ on the join condition $\lambda((R, S))$.

For the following, we will assume a family of hash functions $h = \{h^{(1)}, h^{(2)}, \ldots\}$, where the hash function $h^{(i)}$ takes $i$ attribute values as input and returns a value from $\mathcal{N}$; we will simply use the notation $h$ when the context is clear.

We can now define recursively the distribution scheme $\mathbf{P}_{\mathbb{G}}^{h}$ of the instance $I$ as specified by the co-hash tree $\mathbb{G}$ and the hash family $h$. For every tuple $t \in R^I$ we define:

$$\mathbf{P}_{\mathbb{G}}^{h}(t, I) := \begin{cases} h(t[\alpha(R)_1], t[\alpha(R)_2], \ldots), & \text{if } \mathcal{J}_{\uparrow}^{\mathbb{G}}(t, I) = \emptyset \\ \bigcup_{s \in \mathcal{J}_{\uparrow}^{\mathbb{G}}(t,I)} \mathbf{P}_{\mathbb{G}}^{h}(s, I), & \text{otherwise.} \end{cases}$$

In other words, if a tuple $t$ has an empty upwards join set, then it is hash-partitioned using the attributes in $\alpha(R)$. For simplicity of notation, we will often write $\alpha(t)$ to denote the vector $\langle t[\alpha(R)_1], t[\alpha(R)_2], \ldots \rangle$. If the upwards join set is not empty, then it is collocated with every tuple in $\mathcal{J}_{\uparrow}^{\mathbb{G}}(t, I)$. The data distribution scheme is always well-defined, since we require that the co-hash graph is a collection of disjoint rooted in-trees.

Given a co-hash graph $\mathbb{G}$, we denote by $\mathcal{P}_{\mathbb{G}} = \{\mathbf{P}_{\mathbb{G}}^{h}\}_h$ the family of all distribution schemes $\mathbf{P}_{\mathbb{G}}^{h}$, parameterized by all possible hash functions.

▶ **Example 14.** Figure 1c depicts the induced data distribution on our example instance.

Since $R$ is the root of the co-hash tree, the tuples of $R$ will be hashed using attribute $\langle A_1 \rangle$. For example, $R(1, 2), R(1, 3)$ will always end up in the same node (color red). The tuples from $S$ will be co-hashed according to the join condition $R.A_2 = S.B_1$. For instance, the tuple $S(3, 5)$ joins with two tuples from $R$, $R(1, 3)$ and $R(2, 3)$ and thus will be assigned to where $R(1, 3)$ is (red), and where $R(2, 3)$ is (teal). On the other hand, the tuple $S(8, 9)$ does not join with any tuple from $R$, and hence it will be hashed on attribute $\langle B_1 \rangle$ (magenta).

In the case where $\mathbb{G}$ has no edges, the resulting data distribution reduces to a hash-partitioning strategy, where each relation is distributed independently according to a subset of its attributes.

## 4.2 Practical Considerations

Our definition of a co-hash distribution scheme is based on the concept of *predicate-based reference partitioning* [23]. In the most general version, the join condition $\lambda(e)$ can be any predicate. However, as in [23], we restrict our study to equi-joins, since $(i)$ they cover almost all practical scenarios, $(ii)$ other join conditions (e.g., disequality, inequality) may lead to very large replication. We should note here a fundamental difference with [23]. If a tuple in a non-root relation does not join with any tuple from its parent relation, we make sure that the tuple is hashed. In predicate-based reference partitioning, such a tuple is instead arbitrarily distributed to a node. However, this can hurt data locality, as the next example shows.

▶ **Example 15.** Consider the query $q(x, y) = S(x, y), U(x)$. As we will see in Section 6 $q$ is p-correct for the co-hash graph in Figure 1a. On the other-hand, if we do not specify explicitly that the non-joining tuples of $R, U$ are hashed according to $B_1, D_1$ respectively, the query would not be p-correct. To see this, consider the instance $\{S(a, b), U(a), R(c, d)\}$. Note that the tuples $S(a, b), U(a)$ satisfy $q$, but since neither of the two joins with $R$, they will be assigned to arbitrary nodes.

The declarative framework of distribution constraints introduced in [11] also captures predicate-based reference partitioning, but it cannot control how the non-joining tuples are assigned to nodes as we do in this paper. As we show in the above example, this limitation means that fewer queries may be p-correct for a given scheme.

## 4.3 Some Useful Notions

We next discuss notions and properties of co-hash graphs that will be used throughout the paper.

▶ **Definition 16** (Terminating Path). *Let $\mathbb{G}$ be a co-hash graph, and $I$ an instance. The tuple sequence $t_0 \to t_1 \to \cdots \to t_k$ is a* terminating path *for $t_0 \in I$ if $(i)$ for every $j = 0, \ldots, k-1$ we have $t_{j+1} \in \mathcal{J}_{\uparrow}^{\mathbb{G}}(t_j, I)$, and $(ii)$ $\mathcal{J}_{\uparrow}^{\mathbb{G}}(t_k, I) = \emptyset$.*

Note that the above terminating path for $t_0$ implies that $t_0$ will be assigned to node $h(\alpha(t_k))$ for the scheme $\mathbf{P}_{\mathbb{G}}^{h}$.

We also define a *hash destination* to be a vector of values that is passed to the hash function $h$. For a tuple $t \in I$, $H(t, I)$ is the set of hash destinations that is used to assign tuple $t$ to a node. For instance, if $t \in R^I$ and $R$ is a root node in $\mathbb{G}$ with $\alpha(R) = \langle A, B\rangle$, then $H(t, I) = \{\langle t[A], t[B]\rangle\}$.

**Tuple Collocation.** Given a co-hash graph $\mathbb{G}$, an instance $I$ and a set of tuples $S \subseteq I$, we write $I \rhd_{\mathbb{G}} S$ if for every $\mathbf{P} \in \mathcal{P}_{\mathbb{G}}$ we have $\bigcap_{s \in S} \mathbf{P}(s, I) \neq \emptyset$. In other words, $I \rhd_{\mathbb{G}} S$ if the tuples from $S$ are always collocated in some common node, no matter the choice of the hash family. We show next that we can decide this problem in P. Intuitively, this holds because of the acyclic structure of the co-hash graph.

▶ **Lemma 17.** *Let $\mathbb{G} = (V, E, \alpha, \lambda)$ be a co-hash graph, $I$ be an instance, and $S \subseteq I$. Then, we can decide in polynomial time whether $I \rhd_{\mathbb{G}} S$.*

## 5 Properties of Co-hashing

### 5.1 Balancedness

We first discuss how co-hash schemes can overcome the lower bound for oblivious schemes.

▶ **Example 18.** Consider again the query we used in Example 11, $q(x, y, z, w) = R(x, y), S(y, z), T(z, w)$. Suppose that the relations $R, S, T$ are distributed according to the co-hash graph of Figure 1a. It is easy to see that $q$ is parallel-correct.

We now turn into analyzing the balancedness of the distribution scheme induced by the co-hash graph. Suppose that $A_1$ is a key in $R$, $B_1$ is a key in $S$, and $C_1$ is a key in $T$; this is a common scenario, since many joins in practice are key-foreign key joins. In this case, assuming that $|I| \gg p$, one can show that the balancedness will be $b = O(1)$, which is asymptotically close to the best possible value of 1. On the other hand, as we showed in the previous section, any oblivious scheme will have balancedness $\Omega(p^{1/2})$.

Our first result generalizes the above example. We show that if the hash signatures and join conditions involve keys of the relations, balancedness is guaranteed to be constant. This result captures a lot of real-world examples, since in practice co-hash graphs are constructed by following the key-foreign key constraints of the schema.

▶ **Lemma 19.** *Let $\mathbb{G}$ be a co-hash graph such that $(i)$ for every edge $e = (R, S)$, the attributes in $\lambda(e)$ form a superkey for $R$, and $(ii)$ for every vertex $R$, the attributes in $\alpha(R)$ form a superkey for $R$. Let $I$ be an instance such that $|I| \gg p$. Then, $b(\mathcal{P}_{\mathbb{G}}, I) = O(1)$.*

Our second result shows that any (non-trivial) co-hash partitioning scheme has constant balancedness when the input database has bounded degree. Formally, a *bounded degree* database is one where the number of times each value appears in a tuple is bounded by a constant $d \in \mathbb{N}$. This is not a surprising result, since the constraint of bounded degree means that each tuple can join with at most $d$ other tuples. Contrast this result with oblivious schemes, where even for bounded degree instances the balancedness is a non-constant function of the number of nodes $p$.

▶ **Lemma 20.** *Let $\mathbb{G}$ be a co-hash graph such that $(i)$ no join conditions are empty, and $(ii)$ every hash signature is of size at least 1. Let $I$ be a bounded degree instance with bound $d$ such that $|I| \gg p$. Then $b(\mathcal{P}_{\mathbb{G}}, I) = O(d^{\ell})$, where $\ell$ is the maximum height of a rooted in-tree in $\mathbb{G}$.*

### 5.2 Obliviousness

Let $\Sigma$ be a database schema and $\mathbb{G}$ a co-hash graph over $\Sigma$. The question we ask here is: which relations in $\Sigma$ are oblivious w.r.t. $\mathcal{P}_{\mathbb{G}}$?

---

CoHash-Oblivious

| | |
|---|---|
| **Input:** | co-hash graph $\mathbb{G}$, relation $R$ |
| **Question:** | Is $R$ oblivious with respect to $\mathcal{P}_{\mathbb{G}}$? |

---

Before we describe how we can decide the above property, we need some additional technical machinery.

▶ **Definition 21** (Hash-Compatible). *Let $\mathbb{G} = (V, E, \alpha, \lambda)$ be a co-hash graph, and $R, S \in V$. We say that $R$ is* hash-compatible *with $S$ w.r.t. an equivalence relation $\rho$ over* att, *denoted $\rho \models R \parallel S$, if:*

1. *$\alpha(R), \alpha(S)$ have the same arity; and*
2. *for every position $i$, $\alpha(R)_i =_\rho \alpha(S)_i$.*

▶ **Example 22.** Consider the co-hash tree rooted at $R$ in the co-hash graph presented in Figure 1a. Consider relations $R$ and $S$, and let $\rho = \{A_1 = B_1, A_2 = B_1\}$.

We claim that $\rho^\oplus \models R \parallel S$. Indeed, $|\alpha(R)| = |\alpha(S)| = 1$, so the arities of $R$ and $S$ are the same. Also, $\alpha(R)_1 = A_1$, $\alpha(S)_1 = B_1$, and $A_1 = B_1 \in \rho^\oplus$. Hence, $\alpha(R)_1 =_{\rho^\oplus} \alpha(S)_1$.

On the other hand, if $\rho = \{A_2 = B_1\}$, then $\alpha(R)_1 \neq_{\rho^\oplus} \alpha(S)_1$, and $\rho^\oplus \models R \parallel S$ does not hold.

▶ **Lemma 23.** *Let $\mathbb{G} = (V, E, \alpha, \lambda)$ be a co-hash graph, and $R \in V$. Let $(R =)S_0 \xrightarrow{e_1} S_1 \xrightarrow{e_2} \dots \xrightarrow{e_k} S_k$ be the (unique) root path of $R$. If $(\lambda(e_1) \cup \dots \cup \lambda(e_j))^\oplus \models R \parallel S_j$ for every node $j = 1, \dots, k$, then for every hash family $h$ and every instance $I$ we have $\mathbf{P}^h_{\mathbb{G}}(t, I) = \{h(\alpha(t))\}$.*

We can now state the main theorem of this section.

▶ **Theorem 2.** *Let $\mathbb{G} = (V, E, \alpha, \lambda)$ be a co-hash graph, and $R \in V$. Let $(R =)S_0 \xrightarrow{e_1} S_1 \xrightarrow{e_2} \dots \xrightarrow{e_k} S_k$ be the root path of $R$. The following are equivalent:*

1. *$\mathcal{P}_{\mathbb{G}}$ is oblivious for $R$.*
2. *$R$ is hashed on $\alpha(R)$.*
3. *For every $i = 1, \dots, k$, we have $\rho_i \models R \parallel S_i$, where $\rho_i = (\lambda(e_1) \cup \dots \cup \lambda(e_i))^\oplus$ .*

▶ **Example 24.** Consider the co-hash tree rooted at $V$ in the co-hash graph presented in Figure 1a. Consider relation $X$ in the co-hash tree. The root path of $X$ is $X \to V$. We have $\rho_1 = \{G_1 = E_1\}$, which means that $\rho_1 \models R \parallel V$ holds. Hence, condition (3) of Theorem 2 holds and $X$ is oblivious.

▶ **Proposition 1.** *CoHash-Oblivious is in* P.

## 5.3 Clustering

In this section we consider whether a co-hash graph $\mathbb{G}$ induces a clustering of a relation in the schema. Recall that by checking whether a relation is clustered, we implicitly also check about the presence of redundancy.

| CoHash-Clustered | |
|---|---|
| **Input:** | co-hash graph $\mathbb{G}$, relation $R$, $\mathbf{A} \subseteq \mathsf{att}(R)$ |
| **Question:** | Is $R$ $\mathbf{A}$-clustered w.r.t. $\mathcal{P}_{\mathbb{G}}$? |

**Algorithm 1** CHECKING CLUSTERING.

```
1: **Input**: $\mathbb{G} = (V, E, \alpha, \lambda)$, $R \in V$, $\mathbf{A} \subseteq \mathsf{att}(R)$
2: Let root path $(R =)S_0 \xrightarrow{e_1} S_1 \xrightarrow{e_2} \dots \xrightarrow{e_k} S_k$
3: $\mathcal{V}_0 \leftarrow \mathbf{A}^+$
4: $i \leftarrow 0$
5: while $i < k$ do
6:     if $\exists j$ s.t. $\alpha(S_i)_j \notin \mathcal{V}_i$ then                    ▷ (VC)
7:         return false
8:     if $\exists (A, B) \in \lambda(e_{i+1})$ s.t. $A \notin \mathcal{V}_i$ then       ▷ (EC)
9:         break
10:    $i$++
11:    $\mathcal{V}_i \leftarrow \{B \in \mathsf{att}(S_i) \mid \exists A \in \mathcal{V}_{i-1} : (A = B) \in \lambda(e_i)\}^+$
12: if $i = k$ and $\exists j : \alpha(S_k)_j \notin \mathcal{V}_k$ then
13:     return false
14: else
15:     for $j = i + 1, \dots, k$ do
16:         if $(\lambda(e_1) \cup \dots \cup \lambda(e_j))^{\oplus} \models S_i \parallel S_j$ then     ▷ (HC)
17:             return false
18: return true ;
```

An easy observation is that obliviousness implies that $R$ is $\mathsf{att}(R)$-clustered, and so it is non-redundant. Indeed, from Theorem 2, an oblivious relation allocates each tuple to exactly one location, and thus it is always non-redundant. The inverse is not true, as the next example demonstrates.

▶ **Example 25.** Consider two relations, $R(A_1, A_2)$, $S(B_1, B_2)$, and a co-hash graph $\mathbb{G} = (V, E, \alpha, \lambda)$ with $V = \{R, S\}$, $E = \{(S, R)\}$ and $\lambda((S, R)) = \{B_1 = A_1\}$. Moreover, let $\alpha(S) = \langle B_2 \rangle$ and $\alpha(R) = \langle A_1 \rangle$.

Observe first that $S$ is not oblivious, since $S$ is not hash-compatible with $R$ w.r.t. $\{B_1 = A_1\}^{\oplus}$. We will argue next that $S$ is non-redundant. Consider any instance $I$ and a tuple $s \in S^I$. If $s$ does not join with any tuple in $R^I$ on $A_1 = B_1$, then it is simply hashed on $\langle B_2 \rangle$ and is non-redundant. So suppose that $s$ joins with tuples $\{r_1, \dots, r_n\}$ in $R^I$. Hence, $s$ will end up in the nodes $\{h(r_1[A_1]), \dots, h(r_n[A_1])\}$. However, for any two tuples $r_i, r_j$, we have $r_i[A_1] = s[B_1] = r_j[A_1]$, and hence $h(r_1[A_1]) = h(r_2[A_1]) = \dots = h(r_n[A_1])$, which means that $s$ will be assigned to exactly one node.

A second observation is that functional dependencies are now critical in deciding whether there is redundancy (and in general clustering) in a relation or not.

▶ **Example 26.** Consider the same two relations $R(A_1, A_2)$ and $S(B_1, B_2)$, and a co-hash graph $\mathbb{G} = (V, E, \alpha, \lambda)$ with $V = \{R, S\}$, $E = \{(S, R)\}$, $\lambda((S, R)) = \{B_1 = A_1\}$. Moreover, let $\alpha(S) = \langle B_2 \rangle$ and $\alpha(R) = \langle A_2 \rangle$.

In contrast to the previous example, where the hash signature of $R$ was $\langle A_1 \rangle$, $S$ is now redundant. However, if we add to relation $R$ the functional dependency $A_1 \to A_2$, $S$ becomes non-redundant.

We now present an algorithm (Algorithm 1) that decides in polynomial time whether a relation $R$ is $\mathbf{A}$-clustered for some $\mathbf{A} \subseteq \mathsf{att}(R)$ (and hence also decides non-redundancy).

The algorithm takes as input the co-hash graph $\mathbb{G}$, the relation $R$, and an attribute set $\mathbf{A} \subseteq \mathsf{att}(R)$. The algorithm starts from the node $R$, and then traverses the root path of $R$ bottom up. At each relation $S_i$ of the root path, it computes inductively a set of attributes

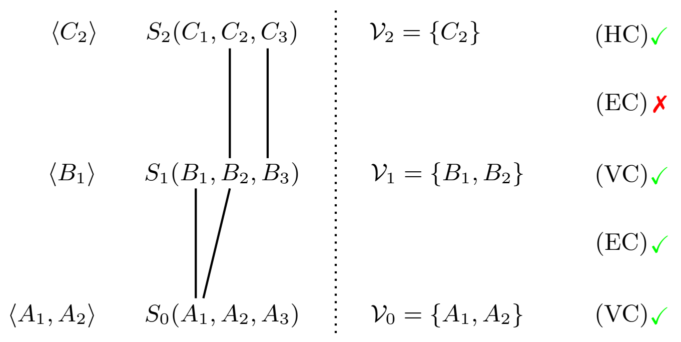


**Figure 2** Example execution of Algorithm 1 to check whether relation $S_0$ is $\mathbf{A}$-clustered, where $\mathbf{A} = \{A_1, A_2\}$. The edges between attributes depict the join conditions.

$\mathcal{V}_i \subseteq \mathsf{att}(S_i)$. While computing each $\mathcal{V}_i$, we also take into account functional dependencies. It then uses this set to check two conditions: one condition (EC) on each edge of the root path, and one condition (VC) on each vertex of the root path. If the vertex condition fails, the algorithm returns false. Otherwise, if the edge condition (EC) is not satisfied, then we check the remaining path for hash compatibility (HC).

We illustrate the working of Algorithm 1 in an example:

▶ **Example 27.** Consider the co-hash graph depicted in Figure 2. Suppose there are no functional dependencies between attributes in any of the relations. We will run the algorithm to check whether $S_0$ is $\mathbf{A}$-clustered, where $\mathbf{A} = \{A_1, A_2\}$. We initially have $\mathcal{V}_0 = \mathbf{A}^+ = \{A_1, A_2\}$. The first vertex condition (VC) for $i = 0$ is satisfied, since both attributes in the hash signature of $S_0$ are in $\mathcal{V}_0$. The first edge condition (EC) is also satisfied, since the join condition between $S_0, S_1$ only uses attribute $A_1$, which belongs in $\mathcal{V}_0$. Similarly, one can check that the second (VC) is also satisfied. However, the second (EC) for $i = 1$ fails, since $B_3 = C_3 \in \lambda(e_2)$, but $B_3 \notin \mathcal{V}_1$. The algorithm now enters the loop in lines 15-17, where it checks for hash compatibility between $S_1$ and all the relations up to the root. In this example, we need to check only one (HC), between $S_1$ and $S_2$. Since $B_1 = C_2 \in (\lambda(e_1) \cup \lambda(e_2))^{\oplus}$, hash compatibility holds, and thus the algorithm returns true. Thus, $S_0$ is $\{A_1, A_2\}$-clustered. This also implies that $S_0$ is $\{A_1, A_2, A_3\}$-clustered, so also non-redundant.

▶ **Theorem 3.** *Let $\mathbb{G} = (V, E, \alpha, \lambda)$ be a co-hash graph, relation $R \in V$, and $\mathbf{A} \subseteq \mathsf{att}(R)$. Algorithm 1 runs in polynomial time (in the size of $\mathbb{G}$) and returns true if and only if $R$ is $\mathbf{A}$-clustered w.r.t. $\mathcal{P}_{\mathbb{G}}$. Hence,* CoHash-Clustered *is in* P.

## 6 Parallel Correctness

In this section, we investigate the problem of p-correctness for data distributions induced by a co-hash graph. We start by considering the following decision problem, where we check for p-correctness for an instance that is given as input to the problem.

| I-CPC | |
|---|---|
| **Input:** | co-hash graph $\mathbb{G}$, input $I$, query $q$ |
| **Question:** | Is $q$ p-correct on $I$ under $\mathcal{P}_{\mathbb{G}}$? |

For CQs, the above problem is $\Pi_2^P$-complete. Interestingly, the hardness result comes from the observation that a co-hash scheme can implicitly encode an arbitrary oblivious distribution scheme. More precisely, for each relation $R$, the scheme is encoded as another relation $R'$ that stores the node destinations for each tuple in $R$. However, we can obtain a better complexity result if we restrict to full CQs, which correspond to joins without projections.

▶ **Theorem 4.** *I-CPC is $\Pi_2^P$-complete for CQs.*

▶ **Theorem 5.** *I-CPC is* coNP*-complete for full CQs.*

We next consider the problem of p-correctness across all possible instances.

CPC

| | |
|---|---|
| **Input:** | co-hash graph $\mathbb{G}$, query $q$ |
| **Question:** | Is $q$ p-correct under $\mathcal{P}_{\mathbb{G}}$? |

The next example shows that deciding p-correctness is not always straightforward.

▶ **Example 28.** Consider the co-hash graph in Figure 1a, and the queries $q_1(x, y, z) = R(x, y), S(y, z)$ and $q_2(x, y, z, w) = R(x, y), S(y, z), T(z, w)$. Both queries can be computed locally without any data shuffling, since by construction of the co-hash distribution, tuples that join are placed in the same node. Hence, both are p-correct. However, there are other queries that are p-correct in more indirect ways. For instance, the query $q_3(x, y) = S(x, y), U(x)$ is also p-correct.

Our first result is that the above problem is NP-hard for general CQs. The hardness comes from a reduction from the problem of query containment. It is worth noting that the lower bound holds even for co-hash graphs with no edges, which correspond to oblivious distribution schemes. On the other hand, for full CQs the decision problem for p-correctness is in polynomial time.

▶ **Theorem 6.** *CPC is* NP*-hard for CQs.*

▶ **Theorem 7.** *CPC is in* P *for full CQs.*

Our polynomial time algorithm generates a polynomial number of small instances with labelled nulls (denoted by $\perp_o$ ) and constants, and then makes a collocation check for each one of the generated instances.

**Functional Dependencies.** Each node $R$ in the co-hash graph is associated with a set of functional dependencies $\mathsf{fd}(R)$. Let $F = \bigcup_R \mathsf{fd}(R)$. We will need to apply the chase algorithm to an instance $I$ w.r.t. $F$, which results in a new instance $I'$ (or the chase fails). It will be convenient to capture the result of the chase by a homomorphism $\theta$ such that $I' = \theta(I)$. We then write $\theta = \mathit{chase}(I, F)$.

**The Extension Step.** We now describe a procedure, called $\mathit{extend}(I, t)$, that takes as input an instance $I$ (with labelled nulls and constants) and a tuple $t \in R^I$. The procedure can fail, in which case it returns $\perp$.

If $R$ has no outgoing edge in $\mathbb{G}$, the procedure fails. Otherwise, let $e = (R, S)$ be the unique outgoing edge from $R$. Intuitively, we will try to extend $I$ with a fresh most general tuple from $S$ that joins with $t$ on $\lambda(e)$.

As a first step, we construct a homomorphism $\zeta$ that maps the labelled nulls of $I$ onto themselves (or constants) such that $\zeta(t[A_i]) = \zeta(t[A_j])$ whenever $A_i = A_j \in \lambda(e)^{\oplus}$ with $i \neq j$, otherwise it is the identity mapping. This step may fail when $t[A_i], t[A_j]$ are distinct constants, in which case the procedure fails. Let $I' = \zeta(I)$ and $t' = \zeta(t)$.

As a second step, we construct a new tuple $s$ from $S$ such that for every $A = B \in \lambda(e)$ with $A \in \mathsf{att}(R), B \in \mathsf{att}(S)$ we have $s[B] = t'[A]$, and for every non-joining attribute we introduce a fresh labelled null that does not occur in $I'$. Because of how we constructed $I'$, the tuple $s$ is always well-defined. Finally, we apply the chase to the instance $I' \cup \{s\}$. If the chase fails, we return $\perp$. Otherwise, let $\chi = \mathit{chase}(I' \cup \{s\}, F)$. The procedure returns the pair $(\chi \circ \zeta, \chi(s))$.

**Algorithm.** Let $D[q]$ be the canonical instance of $q$ (we assume w.l.o.g. that we have chased $q$ w.r.t. $F$). If the tuples in $D[q]$ are not collocated, then $D[q]$ is a witness instance that proves that $q$ is not p-correct. Recall that we can do this check in polynomial time using Lemma 17. However, if the tuples are collocated, then this is not a sufficient condition for p-correctness, and we need to check additional instances. We do this through Algorithm 2. It starts with the canonical instance $D[q]$ of the query $q$, and checks whether the tuples in $D[q]$ are collocated for this instance (Line 2). Then, for every tuple in $D[q]$, it initiates a sequence of extension steps: each step is applied using the tuple generated in the previous step (if possible). The algorithm checks whether the tuples in $D[q]$ are collocated in every instance generated in this fashion; if so, it returns true, otherwise it terminates with false.

**Algorithm 2** Deciding P-Correctness.

```
1:  **Input**: co-hash graph $\mathbb{G}$, full CQ $q$
2:  **if** $D[q] \not\triangleright_{\mathbb{G}} D[q]$ **then**
3:      **return** false ;
4:  **for** each $\tilde{t} \in D[q]$ **do**
5:      $i \leftarrow 0$ ; $D_0 \leftarrow D[q]$ ; $\tilde{s}_0 \leftarrow \tilde{t}$ ; $I_0 \leftarrow D[q]$
6:      **while** $\mathit{extend}(I_i, \tilde{s}_i) \neq \perp$ **do**
7:          $i \leftarrow i + 1$ ;
8:          $(\phi_i, \tilde{s}_i) \leftarrow \mathit{extend}(I_{i-1}, \tilde{s}_{i-1})$ ;
9:          $I_i \leftarrow \phi_i(I_{i-1}) \cup \{\tilde{s}_i\}$ ;
10:         $D_i \leftarrow \phi_i(D_{i-1})$ ;
11:         **if** $I_i \not\triangleright_{\mathbb{G}} D_i$ **then**
12:             **return** false ;
13: **return** true ;
```

We analyze the runtime and correctness of the above algorithm in the appendix. Here, we present two examples of its execution.

▶ **Example 29.** Consider the co-hash graph in Figure 1a, and the query $q_4(x, y, z) = R(x, y), S(z, x)$. The canonical instance for this query is $I_0 = \{R(\perp_x, \perp_y), S(\perp_z, \perp_x)\}$. The hash destination for $R(\perp_x, \perp_y)$ is $\langle \perp_x \rangle$, and for $S(\perp_z, \perp_x)$ it is $\langle \perp_z \rangle$, hence the two tuples are not collocated. This means that the algorithm exits early and outputs false.

▶ **Example 30.** Consider the co-hash graph in Figure 1a, and the query $q_3(x, y) = S(x, y), U(x)$. The canonical instance for this query is $I_0 = \{S(\perp_x, \perp_y), U(\perp_x)\}$. The hash destination for $S(\perp_x, \perp_y)$ is $\langle \perp_x \rangle$, and for $U(\perp_x)$ it is $\langle \perp_x \rangle$ as well, hence the two tuples are always collocated in $I_0$.

The extension step for the tuple $S(\perp_x, \perp_y)$ returns a tuple $R(\perp_w, \perp_x)$ and the identity homomorphism. Hence, $I_1 = I_0 \cup \{R(\perp_w, \perp_x)\}$ and $D_1 = D[q]$. Note that in $I_1$, both $S(\perp_x, \perp_y)$ and $U(\perp_x)$ join with $R(\perp_w, \perp_x)$, hence the hash destination for both is $\langle \perp_w \rangle$.

Hence, the tuples are collocated in $I_1$ as well. Since $R$ is a root relation in the co-hash graph, the next extension step fails and the loop terminates.

The extension step for the tuple $U(\bot_x)$ returns a new tuple $R(\bot_v, \bot_x)$ and the identity homomorphism. Hence, $I_1 = I_0 \cup \{R(\bot_v, \bot_x)\}$ and $D_1 = D[q]$. As before, the two tuples from $D_1$ are collocated. The next extension step fails and thus the loop terminates. The algorithm will now terminate and output that the query is indeed p-correct.

## 7 Parallel Disjoint Correctness

In this section, we study in analogy to the previous section the problem of pd-correctness.

I-CPDC

|  |  |
|---|---|
| **Input:** | co-hash graph $\mathbb{G}$, input $I$, query $q$ |
| **Question:** | Is $q$ pd-correct on $I$ under $\mathcal{P}_{\mathbb{G}}$? |

CPDC

|  |  |
|---|---|
| **Input:** | Co-hash graph $\mathbb{G}$, query $q$ |
| **Question:** | Is $q$ pd-correct under $\mathcal{P}_{\mathbb{G}}$? |

The complexity landscape for pd-correctness follows the same pattern with the corresponding problems for p-correctness.

▶ **Theorem 8.** *I-CPDC is $\Pi_2^P$-complete for CQs and* coNP*-complete for full CQs.*

▶ **Theorem 9.** *CPDC is* NP*-hard for CQs, and in* P *for the class of full CQs.*

▶ **Example 31.** Consider the co-hash graph in Figure 1a, and the p-correct query $q(x, y, z) = R(x, y), S(z, x)$. The dominant atom, $R$, is the non-redundant root. This makes $q$ pd-correct.

Consider another p-correct query $q(x, y) = S(x, y), U(x)$ on the same co-hash graph. Both atoms are dominant, and both are redundant. Hence, the query is not pd-correct. However, if we add to relation $R$ the fd $A_2 \rightarrow A_1$, then both become non-redundant and $q$ will be pd-correct.

## 8 Related Work

There has been a lot of work in studying parallel evaluation of queries. The massively parallel communication (MPC) model was introduced in [14, 15] to analyze multiway joins and to obtain bounds on the amount of communication and synchronization [1, 5]. The case of query evaluation in a single round of communication has been of particular interest, where data is shuffled once before a query is evaluated. The notion of parallel correctness was introduced in [3, 4] to study query evaluation in one round w.r.t. a distribution policy. [13] extended the study of parallel correctness of conjunctive queries to incorporate bag semantics and [9] extended the ideas to unions of conjunctive queries with negation. Distribution policies and parallel correctness results have also been studied in the context of Datalog [12, 18].

The studies mentioned above have focused on *oblivious* distribution policies, where the destination of a tuple is independent of the input, whereas our work focuses on locality-aware schemes. Further, we assume that the partitioning step (the data shuffling phase) is done as a preprocessing step before any query is query evaluated. In effect, we study query evaluation with “zero” rounds of communication. This mode of “partition once, run queries multiple times” is the approach taken by multiple systems that are tuned for OLAP style queries [7, 21, 17]. Recently, Geck et. al [11] introduced a declarative framework that captures constraints of distributed data; these constraints can capture several classes of non-oblivious distribution schemes. The co-hash schemes we describe in this paper cannot be captured by their framework because we explicitly hash tuples that do not join with the parent relation. In addition, we go beyond the instance-independent p-correctness that was examined in [11], and we study several other properties of theoretical and practical interest.

The idea of location-aware partitioning has been deployed in multiple systems. It has been shown to improve query performance drastically for queries that can take advantage of the collocation by reducing data shuffling across nodes [7, 16, 19]. Reference-based partitioning has been proposed in [7] and join-predicate based partitioning in [22, 23]; it is also deployed in [19]. However, none of these proposals study the properties of the resulting partitioning scheme, or how to decide which queries can be executed without any data shuffling. In fact, the query evaluation procedure of [23] ends up shuffling data for queries that could have been evaluated without any data movement.

## 9 Conclusions and Future Work

In this work, we initiate the formal study of co-hash partitioning, a popular locality-aware data distribution scheme. One immediate direction for future work is to extend the study to UCQs (with negation as well), similar to the extensions in [10]. It is also interesting to consider what happens for queries that are not p-correct for a given co-hash graph. In this case, it will be useful to determine the best data shuffling strategy such that the amount of communication and load per node is minimized. Finally, in this paper we study problems when the co-hash graph is given; an orthogonal problem is to design co-hash schemes that are optimized for a particular query workload.

## References

1 Foto N. Afrati, Paraschos Koutris, Dan Suciu, and Jeffrey D. Ullman. Parallel skyline queries. In *ICDT*, pages 274–284, 2012. doi:10.1145/2274576.2274605.

2 Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010. doi:10.1145/1739041.1739056.

3 Tom J. Ameloot, Gaetano Geck, Bas Ketsman, Frank Neven, and Thomas Schwentick. Parallel-correctness and transferability for conjunctive queries. In *PODS*, 2015. doi:10.1145/2745754.2745759.

4 Tom J. Ameloot, Gaetano Geck, Bas Ketsman, Frank Neven, and Thomas Schwentick. Data partitioning for single-round multi-join evaluation in massively parallel systems. *SIGMOD Record*, 45(1):33–40, 2016. doi:10.1145/2949741.2949750.

5 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In *PODS*, pages 273–284, 2013. doi:10.1145/2463664.2465224.

6 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *J. ACM*, 64(6):40:1–40:58, 2017. doi:10.1145/3125644.

7 George Eadon, Eugene Inseok Chong, Shrikanth Shankar, Ananth Raghavan, Jagannathan Srinivasan, and Souripriya Das. Supporting table partitioning by reference in Oracle. In *SIGMOD*, pages 1111–1122, 2008. doi:10.1145/1376616.1376727.

8 Sumit Ganguly, Avi Silberschatz, and Shalom Tsur. Parallel bottom-up processing of Datalog queries. *J. Log. Program.*, 14(1-2):101–126, October 1992. doi:10.1016/0743-1066(92)90048-8.

9 Gaetano Geck, Bas Ketsman, Frank Neven, and Thomas Schwentick. Parallel-correctness and containment for conjunctive queries with union and negation. In *ICDT*, pages 9:1–9:17, 2016. doi:10.4230/LIPIcs.ICDT.2016.9.

10 Gaetano Geck, Bas Ketsman, Frank Neven, and Thomas Schwentick. Parallel-correctness and containment for conjunctive queries with union and negation. *ACM Trans. Comput. Logic*, 20(3):18:1–18:24, June 2019. doi:10.1145/3329120.

11 Gaetano Geck, Frank Neven, and Thomas Schwentick. Distribution constraints: The chase for distributed data. In *ICDT*, pages 13:1–13:19, 2020. doi:10.4230/LIPIcs.ICDT.2020.13.

12 Bas Ketsman, Aws Albarghouthi, and Paraschos Koutris. Distribution policies for Datalog. In *ICDT*, pages 17:1–17:22, 2018. doi:10.4230/LIPIcs.ICDT.2018.17.

13 Bas Ketsman, Frank Neven, and Brecht Vandevoort. Parallel-correctness and transferability for conjunctive queries under bag semantics. In *ICDT*, pages 18:1–18:16, 2018. doi:10.4230/LIPIcs.ICDT.2018.18.

14 Paraschos Koutris and Dan Suciu. Parallel evaluation of conjunctive queries. In *PODS*, pages 223–234, 2011. doi:10.1145/1989284.1989310.

15 Paraschos Koutris and Dan Suciu. A guide to formal analysis of join processing in massively parallel systems. *SIGMOD Record*, 45(4):18–27, 2016. doi:10.1145/3092931.3092934.

16 Yi Lu, Anil Shanbhag, Alekh Jindal, and Samuel Madden. AdaptDB: Adaptive partitioning for distributed joins. *Proc. VLDB Endow.*, 10(5):589–600, January 2017. doi:10.14778/3055540.3055551.

17 Rimma Nehme and Nicolas Bruno. Automated partitioning design in parallel database systems. In *SIGMOD*, pages 1137–1148, 2011. doi:10.1145/1989323.1989444.

18 Frank Neven, Thomas Schwentick, Christopher Spinrath, and Brecht Vandevoort. Parallel-correctness and parallel-boundedness for Datalog programs. In *ICDT*, pages 14:1–14:19, 2019. doi:10.4230/LIPIcs.ICDT.2019.14.

19 Jags Ramnarayan, Barzan Mozafari, Sumedh Wale, Sudhir Menon, Neeraj Kumar, Hemant Bhanawat, Soubhik Chakraborty, Yogesh Mahajan, Rishitesh Mishra, and Kishor Bachhav. SnappyData: A hybrid transactional analytical store built on Spark. In *SIGMOD*, pages 2153–2156, 2016. doi:10.1145/2882903.2899408.

20 Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed SQL database that scales. *Proc. VLDB Endow.*, 6(11):1068–1079, August 2013. doi:10.14778/2536222.2536232.

21 Young-Kyoon Suh, Ahmad Ghazal, Alain Crolotte, and Pekka Kostamaa. A new tool for multi-level partitioning in Teradata. In *CIKM*, pages 2214–2218, 2012. doi:10.1145/2396761.2398604.

22 Khai Q. Tran, Jeffrey F. Naughton, Bruhathi Sundarmurthy, and Dimitris Tsirogiannis. JECB: A join-extension, code-based approach to OLTP data partitioning. In *SIGMOD*, pages 39–50, 2014. doi:10.1145/2588555.2610532.

23 Erfan Zamanian, Carsten Binnig, and Abdallah Salama. Locality-aware partitioning in parallel database systems. In *SIGMOD*, pages 17–30, 2015. doi:10.1145/2723372.2723718.

## A Proofs

**Proof of Lemma 8.** Since $|\mathbf{P}^{(i)}(I)| \leq |I|$ always holds, we have that $b(\mathcal{P}, I) \leq p$. Equality is achieved by the distribution scheme that sends all its data to a single node. For the other direction, for every $\mathbf{P} \in \mathcal{P}$ we have $\sum_i |\mathbf{P}^{(i)}(I)| \geq r(\mathcal{P}, I) \cdot |I|$. Hence, $\max_i |\mathbf{P}^{(i)}(I)| \geq r(\mathcal{P}, I)|I|/p$ and $b(\mathcal{P}, I) \geq r(\mathcal{P}, I)$. ◀

**Proof of Theorem 1.** The key idea is that if $\mathcal{P}$ is oblivious, then it can be turned into a 1-round algorithm for computing $q$ in the MPC model. In the MPC model, computation is performed in rounds, where each round includes local computation followed by a global exchange of data. Initially, the input data $I$ is distributed arbitrarily across $p$ nodes, and at the end of the computation, the query result must be the union of the local results of each node.

The algorithm works as follows: it chooses $\mathbf{P} \sim \mathcal{P}$ uniformly at random, and then sends each tuple $t \in I$ to the node $\mathbf{P}(t)$. Finally, each node $i \in \mathcal{N}$ computes the query $q$ on its local data, $q(\mathbf{P}^{(i)}(I))$. The correctness of the algorithm follows directly from the fact that $q$ is p-correct w.r.t. $\mathcal{P}$. Additionally, the load of the algorithm $L$ in the MPC model is at least $b(\mathcal{P}, I) \cdot |I|/p$.

We now apply the result from [6], which tells us that any 1-round randomized MPC algorithm on $p$ nodes that computes $q$ correctly must have load $L = \Omega(|I|/p^{1/\tau^*(q)})$. ◀

**Proof of Lemma 17.** The polynomial time algorithm first computes for every tuple $t \in I$ the set of hash destinations $H(t, I)$. This set is computed inductively starting from the root(s) of the co-hash graph and moving to the leaves. For a tuple $t$ in a root relation, we have that $H(t, I) = \{\alpha(t)\}$, i.e., the set of hash destinations is a singleton. For any non-root relation, we look at the upwards join set of $t$. If it is not empty, its hash destination is the union of the hash destinations of the tuples in its upwards join set; otherwise, it is $\{\alpha(t)\}$. This process terminates in polynomial time, since $(i)$ the upwards join set can be computed in polynomial time, and $(ii)$ the possible hash destinations are at most $|V| \cdot |I|$, hence polynomially bounded. As a final step, we compute the intersection $\bigcap_{s \in S} H(s, I)$. If the result is nonempty, then for any choice of hash function (and hence any $\mathbf{P} \in \mathcal{P}_{\mathbb{G}}$), the tuples in $S$ will be collocated in some common node. Otherwise, we can always pick a $p$ large enough and a hash function that does not assign all tuples from $S$ to the same node. ◀

**Proof of Lemma 19.** Let $\langle a_1, \ldots, a_k \rangle$ be a hash destination, where $k \geq 1$. The key observation is that $\langle a_1, \ldots, a_k \rangle$ can be the hash destination of $O(|V|)$ tuples, where $V$ is the vertex set of $\mathbb{G}$. Indeed, by induction it follows that at most one tuple per relation can be assigned to $\langle a_1, \ldots, a_k \rangle$. Since each tuple will be assigned to at least one hash destination, there must be at least $\Omega(|I|/|V|)$ hash destinations. Now, observe that the hash function $h$ sends each hash destination to a node independently and uniformly at random. Let $H$ be the number of hash destinations for instance $I$. Since $|I| \gg p$, the expected maximum number of hash destinations assigned to each node will be $O(H/p)$. Finally, observe that $H \leq |I|$ and that each hash destination has $O(|V|)$ tuples. Hence, $b(\mathcal{P}_{\mathbb{G}}, I) = \frac{p}{|I|} O(|I||V|/p) = O(|V|)$. ◀

**Proof of Lemma 20.** Let $\langle a_1, \ldots, a_k \rangle$ be a hash destination, where $k \geq 1$. Our main observation is that $\langle a_1, \ldots, a_k \rangle$ can be the hash destination of $O(d^\ell)$ tuples. Since each tuple will be assigned to at least one hash destination, there must be at least $\Omega(|I|/d^\ell)$ hash destinations. Now, observe that the hash function $h$ sends each hash destination to a node independently and uniformly at random. Let $H$ be the number of hash destinations for instance $I$. Since $|I| \gg p$, the expected maximum number of hash destinations assigned to each node will be $O(H/p)$. Finally, observe that $H \leq |I|$ and that each hash destination has $O(d^\ell)$ tuples. Hence, $b(\mathcal{P}_{\mathbb{G}}, I) = \frac{p}{|I|} O(|I|d^\ell/p) = O(d^\ell)$. ◀

**Proof of Lemma 23.** Let $\alpha(R) = \langle A_1, \ldots, A_r \rangle$. Consider an instance $I$, a tuple $t \in R^I$, and a terminating path $t \to t_1 \to \cdots \to t_i$, where $i \leq k$. If $\alpha(S_i) = \langle B_1, \ldots, B_r \rangle$ is the hash signature of $S_i$, then $t$ is assigned to node $m_h = h(t_i[B_1], \ldots, t_i[B_r])$. But then, since $(\lambda(e_1) \cup \cdots \cup \lambda(e_j))^\oplus \models R \parallel S_i$ holds, it follows that for every position $\ell$, $A_\ell = B_\ell \in (\lambda(e_1) \cup \cdots \cup \lambda(e_j))^\oplus$. This implies that $\forall \ell, t_i[B_\ell] = t[A_\ell]$, which means that $m_h = h(\alpha(t))$. ◀

**Proof of Theorem 2.** For $3 \Rightarrow 2$, direct application of Lemma 23. $2 \Rightarrow 1$ is trivial. For $1 \Rightarrow 3$, let $S_\mu$ be the first relation in the root path such that $\rho_\mu \models R \parallel S_\mu$ does not hold. Let $\alpha(R) = \langle A_1, \ldots, A_r \rangle$, $\alpha(S_\mu) = \langle B_1, \ldots, B_{r'} \rangle$. We construct a tuple $t_j$ in $S_j$ for $j = 0, \ldots, \mu$ as follows. First, we assign to each equivalence class $\mathcal{C}$ in the equivalence relation $\rho_\mu = (\lambda(e_1) \cup \cdots \cup \lambda(e_\mu))^\oplus$ a distinct constant $a_{\mathcal{C}} \in \mathbf{dom}$. Then, we set $t_j[B] = a_{[B]_{\rho_\mu}}$, where $[B]_{\rho_\mu}$ is the equivalence class where $B$ belongs.

We will now construct two instances $I, I'$ such that for some $h$, $\mathbf{P}^h_{\mathbb{G}}(t_0, I) \neq \mathbf{P}^h_{\mathbb{G}}(t_0, I')$: (1) Instance $I$ contains only the tuple $t_0$. In this instance $t_0$ will be assigned to the node $m_h = h(t_0[A_1], \dots, t_0[A_r])$, and (2) Instance $I'$ contains the tuples $\{t_0, t_1, \dots, t_\mu\}$ as constructed above. Observe first that $t_{j+1} \in \mathcal{J}^{\mathbb{G}}_{\uparrow}(t_j, I')$ for every $j = 0, \dots, \mu - 1$. Indeed, if $(C, D) \in \lambda(e_{j+1})$, then by construction $C, D$ belong in the same equivalence class of $\rho_\mu$ and hence we have that $t_j[C] = t_{j+1}[D]$. Thus, in instance $I'$, $t_0$ will be assigned to the node $m'_h = h(t_\mu[B_1], \dots, t_\mu[B_{r'}])$.

The key observation is that, since $\rho_\mu \models R \parallel S_\mu$ does not hold, either $r \neq r'$, or there exists $(A_i, B_i) \notin \rho_\mu$ such that $A_i$ is not in the same equivalence class as $B_i$ (in which case $t_0[A_i] \neq t_\mu[B_i]$). In either case, we can choose an $h$ such that $m_h \neq m'_h$. ◀

**Proof of Theorem 3.** The next two lemmas prove Theorem 3. ◀

▶ **Lemma 32.** *Let $\mathbb{G} = (V, E, \alpha, \lambda)$ be a co-hash graph, relation $R \in V$, and $\mathbf{A} \subseteq \mathsf{att}(R)$. If Algorithm 1 returns true, then $R$ is $\mathbf{A}$-clustered w.r.t. $\mathcal{P}_{\mathbb{G}}$.*

**Proof.** Assume that $i = \mu$ when the algorithm exits the first loop. Consider some instance $I$, and let $T$ be a set of tuples in $R^I$ such that all tuples in $T$ agree on the attributes in $\mathbf{A}$. We will show that all tuples in $T$ are assigned to the same unique machine. We define $J^i(T, I)$ inductively as follows, for $i = 0, \dots, k$: $J^0(T, I) = T$, and for $i > 0$, $J^i(T, I) = \bigcup_{s \in J^{i-1}(T,I)} \mathcal{J}^{\mathbb{G}}_{\uparrow}(s, I)$. It is easy to see that by construction $J^i(T, I) \subseteq S^I_i$. First, we show the following statement using induction: *(S1) for any $i = 0, \dots, \mu$, and any tuples $s, s' \in J^i(T, I)$ we have $s[\mathcal{V}_i] = s'[\mathcal{V}_i]$.*

For the base case, where $i = 0$, by the definition of $T$ we have $s[\mathbf{A}] = s'[\mathbf{A}]$, and hence $s[\mathbf{A}^+] = s'[\mathbf{A}^+]$. Since $\mathcal{V}_0 = \mathbf{A}^+$, we also have $s[\mathcal{V}_0] = s'[\mathcal{V}_0]$. For the inductive step, let $s, s' \in J^i(T, I)$. By the construction of $J^i(T, I)$, there exist $r, r' \in J^{i-1}(T, I)$ such that $s \in \mathcal{J}^{\mathbb{G}}_{\uparrow}(r, I)$ and $s' \in \mathcal{J}^{\mathbb{G}}_{\uparrow}(r', I)$. Define $\mathcal{V}'_i = \{B \in \mathsf{att}(S_i) \mid \exists A \in \mathcal{V}_{i-1} : (A, B) \in \lambda(e_i)\}$. For any $B \in \mathcal{V}'_i$, there exists some $A \in \mathcal{V}_{i-1}$ such that $(A, B) \in \lambda(e_i)$. This implies that $s[B] = r[A]$ and $s'[B] = r'[A]$. Also, by the inductive hypothesis, it holds that $r[A] = r'[A]$. Hence, $s[B] = s'[B]$. Since $\mathcal{V}_i = (\mathcal{V}'_i)^+$, we also have $s[\mathcal{V}_i] = s'[\mathcal{V}_i]$. Second, we show: *(S2) for any $i = 0, \dots, \mu$, and tuples $r, r' \in J^i(T, I)$, we have $\mathcal{J}^{\mathbb{G}}_{\uparrow}(r, I) = \mathcal{J}^{\mathbb{G}}_{\uparrow}(r', I)$.* Indeed, let $s \in \mathcal{J}^{\mathbb{G}}_{\uparrow}(r, I)$. By the edge condition (EC), for every $(A, B) \in \lambda(e_{i+1})$, we have $A \in \mathcal{V}_i$. By (S1), for every $A \in \mathcal{V}_i$ we have $r[A] = r'[A]$. Thus, $s[B] = r'[A]$, which implies that $s$ joins with $r'$ and hence $s' \in \mathcal{J}^{\mathbb{G}}_{\uparrow}(r, I)$ as well.

We now distinguish two cases. $\exists i \in \{1, \dots, \mu\} : J^i(T, I) = \emptyset$. By (S2), all tuples in $T$ are colocated with the tuples in $J^{i-1}(T, I)$. We will show that all the tuples in $J^{i-1}(T, I)$ are assigned to the same node. If $\alpha(S_{i-1}) = \langle B_1, \dots, B_r \rangle$ is the hash signature of $S_{i-1}$, then any tuple $t \in J^{i-1}(T, I)$ is assigned to the node $m_h = h(t[B_1], \dots, t[B_r])$. Some other tuple $t' \in J^{i-1}(T, I)$ is assigned to $m'_h = h(t'[B_1], \dots, t'[B_r])$. By the vertex condition (VC), every $B_j$ is in $\mathcal{V}_{i-1}$, and hence by (S1), $t[B_j] = t'[B_j]$, which implies that $m_h = m'_h$. Hence, all tuples in $T$ are sent to the same unique node.

Otherwise, suppose $t \in T$ is colocated with some tuple $s \in J^\mu(T, I)$. Let $\alpha(S_\mu) = \langle A_1, \dots, A_r \rangle$. Consider any sequence of tuples $t_1 \in S^I_{\mu+1}, t_2 \in S^I_{\mu+2}, \dots, t_i \in S^I_{\mu+i}$, such that $t_{j+1} \in \mathcal{J}^{\mathbb{G}}_{\uparrow}(t_j, I)$ for every $j \in \{0, \dots, i-1\}$ and $\mathcal{J}^{\mathbb{G}}_{\uparrow}(t_i, I) = \emptyset$. If $\alpha(S_{\mu+i}) = \langle B_1, \dots, B_r \rangle$ is the hash signature of $S_{\mu+i}$, then $s$ (and hence $t$) is assigned to node $m_h = h(t_i[B_1], \dots, t_i[B_r])$. Since by condition (HC) it holds that $(\lambda(e_1) \cup \dots \cup \lambda(e_i))^\oplus \models S_\mu \parallel S_i$, it follows that for every position $\ell$, $(A_\ell, B_\ell) \in (\lambda(e_1) \cup \dots \cup \lambda(e_i))^\oplus$. This implies that $\forall \ell, t_i[B_\ell] = s[A_\ell]$. Hence, tuple $t$ is sent to node $m_h = h(s[A_1], \dots, s[A_r])$. If $t' \in T$ is some other tuple colocated with some $s' \in J^\mu(T, I)$, then similarly we can argue that it is sent to node $m'_h = h(s'[A_1], \dots, s'[A_r])$. But since for every attribute $A_j$ we have $A_j \in \mathcal{V}_\mu$, it holds that $s[A_j] = s'[A_j]$ and hence $m_h = m'_h$. ◀

▶ **Lemma 33.** *Let $\mathbb{G} = (V, E, \alpha, \lambda)$ be a co-hash graph, relation $R \in V$, and $\mathbf{A} \subseteq \mathsf{att}(R)$. If Algorithm 1 returns false, then there $h$ such that $R$ is not $\mathbf{A}$-clustered w.r.t. $\mathbf{P}^h_{\mathbb{G}}$.*

**Proof.** We distinguish two cases, depending on whether only (VC) fails, or both (EC) and (HC) fail together. Suppose that the algorithm returns false because (VC) fails for some relation $S_\mu$, $\mu \geq 0$. We then construct an instance $I$ with labelled nulls as follows. First, we assign to each equivalence class $C$ in the equivalence relation $(\lambda(e_1) \cup \ldots \cup \lambda(e_{\mu-1}))^{\oplus}$ two distinct nulls $\perp_C, \perp'_C$. Every relation $S_j$ for $j = 0, \ldots, \mu$ contains two tuples $t_j, t'_j$ such that for any attribute $A \in C$, $t_j[A] = t'_j[A] = \perp_C$ whenever $A \in \mathcal{V}_j$, otherwise $t_j[A] = \perp_C$ and $t'_j[A] = \perp'_C$. All the other relations are empty. It is easy to see that $I$ satisfies the functional dependencies, and also $t_0, t'_0$ agree on $\mathbf{A}$. Moreover, for every $j < \mu$, we have $t_{j+1} \in \mathcal{J}^{\mathbb{G}}_{\uparrow}(t_j, I)$ and $t'_{j+1} \in \mathcal{J}^{\mathbb{G}}_{\uparrow}(t'_j, I)$. Let $\alpha(S_\mu) = \langle B_1, \ldots, B_r \rangle$. Then, tuple $t_0$ is sent to $h(t_\mu[B_1], \ldots, t_\mu[B_r])$ and tuple $t'_0$ to $h(t'_\mu[B_1], \ldots, t'_\mu[B_r])$. Since (VC) failed at $S_\mu$, there exists some attribute $B_k \notin \mathcal{V}_\mu$, in which case $t_\mu[B_k] \neq t'_\mu[B_k]$. Hence, we can always choose a hash function $h$ such that $\mathbf{P}^h_{\mathbb{G}}(t_0, I) \neq \mathbf{P}^h_{\mathbb{G}}(t'_0, I)$.

Suppose that the algorithm returns false because (EC) and (HC) fail together. Let $\mu \geq 0$ be the first index with join condition $\lambda(e_{\mu+1})$ for which (EC) fails. Moreover, let $S_l$, $l > \mu$, be the first relation where condition (HC) fails. We construct an instance $I$ with labelled nulls as follows. First, we assign to each equivalence class $C$ in the equivalence relation $(\lambda(e_1) \cup \ldots \cup \lambda(e_{\ell-1}))^{\oplus}$ two distinct nulls $\perp_C, \perp'_C$. We construct two types of tuples in $I$: (1) tuples $t_j$ in $S_j$ for $j = 0, \ldots, \ell$. For an attribute $B \in C$, we set $t_j[B] = \perp_C$, and (2) tuples $t'_j$ in $S_j$ for $j = 0, \ldots, \mu$. For an attribute $B \in C$, we set $t'_j[B] = \perp_C$ if $B \in \mathcal{V}_j$, otherwise $t'_j[B] = \perp'_C$.

It is easy to see that $t_0, t'_0$ agree on the attributes in $\mathbf{A}$. Moreover, $t_0$ will be colocated with $t_\ell$, and $t'_0$ will be colocated with $t'_\mu$. Now, since condition (EC) failed, there exists $(A, B) \in \lambda(e_{\mu+1})$ such that $A \notin \mathcal{V}_\mu$. Suppose that $A$ belongs in the equivalence class $C$. By our construction, this implies that $t'_\mu[A] = \perp'_C$, but $t_{\mu+1}[B] = \perp_C$. Hence, $\mathcal{J}^{\mathbb{G}}_{\uparrow}(t'_\mu, I) = \emptyset$. Let $\alpha(S_\mu) = \langle A_1, \ldots, A_r \rangle$ and $\alpha(S_\ell) = \langle B_1, \ldots, B'_r \rangle$. Then, $t_0$ is sent to $m_h = h(t_\ell[B_1], \ldots, t_\ell[B'_r])$, and $t'_0$ to $m_h = h(t'_\mu[A_1], \ldots, t'_\mu[A_r])$. Moreover, since (VC) holds for $S_\mu$, every $A_j \in \mathcal{V}_\mu$, and hence $m_h = h(t_\mu[A_1], \ldots, t_\mu[A_r])$. Finally, since $S_\mu, S_\ell$ are not hash-compatible w.r.t. $(\lambda(e_1) \cup \ldots \cup \lambda(e_{\ell-1}))^{\oplus}$, either $r \neq r'$, or there exists a position $j$ such that $(A_j, B_j)$ is not in the equivalence relation, which would imply that $t_\ell[B_j] \neq t_\mu[A_j]$. In both cases, we can pick a hash function $h$ such that $t_0, t'_0$ end up in different nodes. ◀

**Proof of Theorem 4.** To show membership in $\Pi^P_2$, we will show that the complement is in $\Sigma^P_2$. To this end, we will give a polynomial time algorithm with the following property: there exists a valuation $v$ such that for every valuation $v'$ the algorithm returns yes for $(v, v')$ if and only if $q$ is not p-correct on $I$. First, the algorithm checks whether $v, v'$ satisfy $q$. If $v$ does not satisfy $q$, it returns no. If $v'$ does not satisfy $q$, it returns yes. Then, it checks that $v, v'$ agree on the head of $q$, and if not, it returns yes. Finally, it checks whether the tuples $\{v'(\mathbf{x}_1), \ldots, v'(\mathbf{x}_\ell))$ are collocated in instance $I$. If not, it returns yes, otherwise it terminates with no. It is easy to see that all the checks can be done in polynomial time following from Lemma 17.

To show $\Pi^P_2$-hardness, we construct a reduction from the problem PCI($\mathcal{P}_{\text{fin}}$), defined in [3]. In this problem, we are given a CQ $q$, an instance $I$, and a distribution policy $\mathbf{P}$ which is explicitly enumerated as part of the input (i.e., for each tuple in $I$ we know its destination node). Then, it asks whether $q$ is p-correct in $I$ under $\mathbf{P}$.

We construct a co-hash graph $\mathbb{G} = (V, E, \alpha, \lambda)$ as follows. For every relation $R(A_1, \ldots, A_k)$ in $q$, we introduce two nodes in $V$: one is $R$, and the other is a fresh relation $R'(C, A'_1, \ldots, A'_k)$. $E$ contains edges of the form $(R, R')$, where $\lambda((R, R')) = \{A_1 = A'_1, \ldots, A_k = A'_k\}$. Finally, we set $\alpha(R) = \langle\rangle$ and $\alpha(R') = \langle C\rangle$. For this schema, we create an instance $I'$, where for every tuple $t = R(a_1, \dots, a_k) \in I$ we add the tuples $R(a_1, \dots, a_k)$ and $\{R'(\kappa, a_1, \dots, a_k) \mid \kappa \in \mathbf{P}(t)\}$ to $I'$. Intuitively, each relation $R'$ encodes the destinations of the tuples in $R$ according to $\mathbf{P}$.

The reduction is in polynomial time. We now claim that $q$ is p-correct in $I$ under $\mathbf{P}$ if and only if $q$ is p-correct on $I'$ under $\mathcal{P}_{\mathbb{G}}$. Indeed, notice that for some hash function $h$, tuple $t \in I$ gets assigned to $\{h(\kappa) \mid \kappa \in \mathbf{P}(t)\}$. Hence, if for a set of tuples $T \subseteq I$ we have $\kappa \in \bigcap_{t \in T} \mathbf{P}(t)$, then $h(\kappa) \in \bigcap_{t\in T} \mathbf{P}^h_{\mathbb{G}}(t, I')$ for any $h$, which implies $I' \rhd_{\mathbb{G}} T$. If $\bigcap_{t\in T} \mathbf{P}(t) = \emptyset$, then the hash function $h(\kappa) = \kappa$ implies that $I' \not\rhd_{\mathbb{G}} T$. ◀

**Proof of Theorem 5.** We first show membership in coNP. Let $q(\mathbf{y}) = R_1(\mathbf{x}_1), \dots, R_\ell(\mathbf{x}_\ell)$ be a full CQ, and an instance $I$. We will guess a valuation $v$ over the variables of $q$, and then check that $(i)$ for every $i$, $v(\mathbf{x}_i) \in I$, and $(ii)$ $I \rhd_{\mathbb{G}} \{v(\mathbf{x}_1), \dots, v(\mathbf{x}_\ell)\}$. Indeed, any such valuation will be a witness that p-correctness is violated. It remains to show that $(i)$ and $(ii)$ can be done in polynomial time. Indeed, $(i)$ is straightforward. For $(ii)$, we can apply directly Lemma 17.

We show coNP-hardness by reduction from the problem of CQ evaluation. Consider a boolean CQ $q() = R_1(\mathbf{x}_1), \dots, R_\ell(\mathbf{x}_\ell)$, an instance $I$, and the problem that asks whether $q(I)$ is empty or not, which is known to be NP-hard. We construct a full query $q'$ from $q$ as follows: $q'(\dots) = S_1(z_1), R'_1(z_1, \mathbf{x}_1), \dots, S_\ell(z_\ell), R'_\ell(z_\ell, \mathbf{x}_\ell)$, where $\{z_1, \dots, z_\ell\}$ are distinct fresh variables. To construct an instance $I'$, for every tuple $R_i(\mathbf{a}_i) \in I$, we introduce $R'_i(i, \mathbf{a}_i) \in I'$. Note that this means that a tuple in $I$ can create many copies of it in $I'$ (as many as the number of times its relation occurs in $q$). Moreover, for every $i = 1, \dots, \ell$, we add the tuple $S_i(i)$. It is easy to verify that $q(I)$ is true if and only if $q'(I') \neq \emptyset$. Finally, we construct a co-hash graph $\mathbb{G} = (V, E, \alpha, \lambda)$ as follows: $V$ contains all relations in the body of $q'$, $E = \emptyset$, and $\alpha$ uses only the first attribute of each relation for hashing. Note that $\mathcal{P}_{\mathbb{G}}$ is an oblivious distribution scheme!

We now show the following: $q'$ is p-correct on $I'$ under $\mathcal{P}_{\mathbb{G}}$ if and only if $q(I)$ is false. For the one direction, suppose $q(I)$ is false. Then, $q'(I') = \emptyset$. Hence, no matter how $\mathcal{P}_{\mathbb{G}}$ distributes $I'$, we trivially have p-correctness. For the other direction, suppose $q(I)$ is true. Then, there exists a valuation $v$ over the variables of $q$ that makes it true. This can be extended to a valuation $v'$, where $v'(z_i) = i$, that makes $q'$ true as well. Let $t_1, t_2, \dots, t_\ell \in I'$ be the tuples that correspond to $v'$ for $R'_1, \dots, R'_\ell$. These are all different (since their first attribute must be different). Moreover, $t_i$ is assigned to $h(i)$. This means we can choose an $h$ such that every $t_i$ goes to a different node, which shows that $q'$ is not p-correct. ◀

**Proof of Theorem 6.** We show a reduction from the problem of query containment: given as input two CQs $q_1, q_2$, is $q_1$ contained in $q_2$? Without any loss of generality, we assume that both queries are boolean and use one binary relation $R(A, B)$.

We construct a boolean CQ $q$ as input to CPC as follows. The schema $\Sigma$ contains two relations, $R'(C, A, B)$ and $T(D)$. For every atom $R(x, y)$ in $q_1$, $q$ contains the atom $R'(w^{(1)}, x^{(1)}, y^{(1)})$, and for every atom $R(x, y)$ in $q_2$ it contains the atom $R'(w^{(2)}, x^{(2)}, y^{(2)})$. Moreover, we add an atom $T(w^{(1)})$. Let $q'_1$ ($q'_2$ resp.) be the subquery of $q$ that contains atoms with variable $w^{(1)}$ ($w^{(2)}$ resp.). The input co-hash graph is simple: the hash signature for $R'$ is $\langle C\rangle$ and for $T$ is $\langle D\rangle$, and it contains no edges.

We now claim that $q$ is p-correct under $\mathcal{P}_{\mathbb{G}}$ if and only if $q_1 \subseteq q_2$. For the one direction, assume that $q_1 \subseteq q_2$. Then, it is easy to see that $q$ is equivalent to $q'_1$. Consider any valuation $v$ that satisfies $q'_1$: then all the relevant facts will end up in the same hash destination $h(v(w^{(1)}))$, making the query p-correct. For the opposite direction, assume that $q_1 \not\subseteq q_2$. Let $I$ be the canonical database of $q$, so $q(I)$ is true. By our construction, we have $D[q] = D[q'_1] \cup D[q'_2]$

and $D[q_1'] \cap D[q_2'] = \emptyset$. The facts of $D[q_1']$ will end up on node $\kappa_1 = h(w^{(1)})$, while the facts of $D[q_2']$ on node $\kappa_2 = h(w^{(2)})$. Additionally, we can always choose the hash function $h$ such that $\kappa_1 \neq \kappa_2$. Since $D[q_2']$ does not have any $T$-facts, we have that $q(D[q_2'])$ is false. Since $q_1 \not\subseteq q_2$, we also have that $q(D[q_1'])$ is false. Thus, $I$ falsifies the p-correctness property. ◀

**Proof of Theorem 7.** To analyze the runtime, we first note that the canonical instance has at most as many tuples as the number of atoms $\ell$ in the body of the query. Moreover, for every such tuple, the while loop will terminate after at most $k$ iterations, where $k$ is the length of the longest root path in $\mathbb{G}$ ($k \leq |E|$). This means that the size of any instance $I_i$ generated by the algorithm is always polynomially bounded (in fact, it will be at most $|E| + \ell$). The check $I_i \triangleright_{\mathbb{G}} D_i$ can be done in polynomial time using Lemma 17, while the extension step also runs in polynomial time.

For correctness, we claim that Algorithm 2 outputs true if and only if the query $q$ is p-correct under $\mathcal{P}_{\mathbb{G}}$. The one direction is straightforward. Indeed, if there exists an instance $I_i$ such that $D_i \subseteq I_i$ and the tuples in $D_i$ are not collocated, then $I_i$ is a witness that p-correctness is violated (since any $D_i$ forms an answer for $q(I_i)$).

The other direction is more involved. Consider an instance $I$ and any valuation $v$ such that $t_i = v(\mathbf{x}_i) \in I$ for every $i = 1, \dots, \ell$. Let $T = \{t_1, t_2, \dots, t_\ell\}$. To prove that the query $q$ is p-correct, it suffices to show that $I \triangleright_{\mathbb{G}} T$. Let $T_\Box \subseteq T$ be the set of tuples such that none of their terminating paths go through a tuple in $T$. It is easy to check that $T_\Box \neq \emptyset$. It now suffices to show that $I \triangleright_{\mathbb{G}} T_\Box$.

If there exists a tuple $s \in I$ such that every $t \in T_\Box$ has a terminating path that goes through $s$, then the claim $I \triangleright_{\mathbb{G}} T_\Box$ follows directly. For the remainder of the proof, we assume that this is not the case. Since $D[q]$ is the canonical instance for $q$, there exists a strong[1] homomorphism $\xi : D[q] \to T$. Since the homomorphism is strong, for every $t \in T$ there exists a tuple $\tilde{t} \in D[q]$ such that $\xi(\tilde{t}) = t$. Note that all tuples in the set $\{\tilde{t} \mid t \in T_\Box\}$ must have an empty upwards join set in $D[q]$. Since $D[q] \triangleright_{\mathbb{G}} D[q]$, it must be that all $\alpha(\tilde{t})$ are equal to the same vector $\beta$ (consisting of labelled nulls and constants) for every $t \in T_\Box$. But then, $\alpha(t) = \alpha(\xi(\tilde{t})) = \xi(\beta)$. Consider any $t \in T_\Box$ and a terminating path $(t =)s_0 \to s_1 \to \cdots \to s_k$ in $I$. By our construction, none of the tuples $s_1, \dots, s_k$ are in $T$. We will show that $\alpha(s_k) = \xi(\beta)$; this implies that all tuples in $T_\Box$ are collocated in $h(\xi(\beta))$ for any hash function $h$, hence proving the claim. We start with two observations: (1) The strong homomorphism $\xi$ can be extended to a strong homomorphism from the instance $I_k$ generated by the algorithm for the while loop of $\tilde{t}$ to the instance $J = T \cup \{s_1, \dots, s_k\}$, and (2) Let $\phi = \phi_1 \circ \phi_2 \cdots \circ \phi_k$. Then, for any hash destination $\gamma$ (over labelled nulls and constants) $\xi(\phi(\gamma)) = \xi(\gamma)$.

Note that some tuple $u \in T_\Box$ must have an empty upwards join set in $J$; otherwise, every tuple in $T_\Box$ has a terminating path that goes through $s_k$, a contradiction. Hence, the tuple $\tilde{u} \in I_k$ with $\xi(\tilde{u}) = u$ has an empty upwards join set in $I_k$, which implies that $\alpha(\tilde{u}) = \phi(\beta)$ is its only hash destination. We can also see that $\tilde{t}$ has a unique hash destination in $I_k$, $\alpha(\tilde{s}_k)$. Since $\tilde{u}, \tilde{t}$ must be collocated to guarantee p-correctness for the instance $I_k$, it must be that $\alpha(\tilde{s}_k) = \alpha(\tilde{u}) = \phi(\beta)$. Thus, $\alpha(s_k) = \xi(\phi(\beta)) = \xi(\beta)$, where the last equality is implied by the second observation. ◀

**Proof of Theorem 8.** To show membership in $\Pi_2^P$, we will show that the complement is in $\Sigma_2^P$. To this end, we will give a polynomial time algorithm with the following property: there exists a valuation $v$ such that for every valuation $v'$ the algorithm returns yes for $(v, v')$ if

[1] A homomorphism $h : I \to I'$ is strong if for every tuple $t \in I'$, there exists a tuple $s \in I$ such that $t = h(s)$.

and only if $q$ is not p-correct on $I$. First, the algorithm checks whether $v, v'$ satisfy $q$. If $v$ does not satisfy $q$, it returns no. If $v'$ does not satisfy $q$, it returns yes. Then, it checks that $v, v'$ agree on the head of $q$, and if not, it returns yes. Finally, it checks whether for some hash family $h$, $|\bigcap_i \mathbf{P}^h_{\mathbb{G}}(v'(\mathbf{x}_i), I)| \neq 1$. If not, it returns yes, otherwise it terminates with no. It is easy to see that all the checks can be done in polynomial time from Lemma 17. Indeed, we can modify the construction in Lemma 17 so that is computes the intersection of the hash destinations as well (instead of only checking for emptiness).

To show $\Pi_2^P$-hardness, we first notice that the hardness proof for the problem PCI($\mathcal{P}_{\text{fin}}$) in [3] can be modified to show $\Pi_2^P$-hardness for the pd-correct variant of the problem. Indeed, all p-correct instances for the proof in [3] are also pd-correct. Formally, we define PDCI($\mathcal{P}_{\text{fin}}$) as follows: we are given a CQ $q$, an instance $I$, and a distribution policy $\mathbf{P}$ which is explicitly enumerated as part of the input (i.e., for each tuple in $I$ we know its destination node). Then, it asks whether $q$ is pd-correct in $I$ under $\mathbf{P}$. The reduction is exactly the same as the one in Theorem 4. We construct a co-hash graph $\mathbb{G} = (V, E, \alpha, \lambda)$ as follows. For every relation $R(A_1, \dots, A_k)$ in $q$, we introduce two nodes in $V$: one is $R$, and the other is a fresh relation $R'(C, A'_1, \dots, A'_k)$. $E$ contains edges of the form $(R, R')$, where $\lambda((R, R')) = \{A_1 = A'_1, \dots, A_k = A'_k\}$. Finally, we set $\alpha(R) = \langle\rangle$ and $\alpha(R') = \langle C \rangle$.

For this schema, we create an instance $I'$, where for every tuple $t = R(a_1, \dots, a_k) \in I$ we add the tuples $R(a_1, \dots, a_k)$ and $\{R'(\kappa, a_1, \dots, a_k) \mid \kappa \in \mathbf{P}(t)\}$ to $I'$. Intuitively, each relation $R'$ encodes the destinations of the tuples in $R$ according to $\mathbf{P}$. We now claim that $q$ is pd-correct in $I$ under $\mathbf{P}$ if and only if $q$ is pd-correct on $I'$ under $\mathcal{P}_{\mathbb{G}}$. Indeed, notice that for some hash function $h$, tuple $t \in I$ gets assigned to $\{h(\kappa) \mid \kappa \in \mathbf{P}(t)\}$. Hence, if for a set of tuples $T \subseteq I$ we have $\kappa \in \bigcap_{t \in T} \mathbf{P}(t)$, then $h(\kappa) \in \bigcap_{t \in T} \mathbf{P}^h_{\mathbb{G}}(t, I')$ for any $h$. The result follows by picking $h(\kappa) = \kappa$.

The coNP-hardness proof follows the same structure as the one for p-correctness (Theorem 5). Indeed, in the reduction p-correctness holds only if the query result is empty, in which case pd-correctness also trivially holds. To show membership in coNP, consider a full query $q(\mathbf{y}) = R_1(\mathbf{x}_1), \dots, R_\ell(\mathbf{x}_\ell)$ and an instance $I$. We guess a valuation $v$ over the variables of $q$, and then check that $(i)$ for every $i$, $v(\mathbf{x}_i) \in I$, $(ii)$ for some hash family $h$, $|\bigcap_i \mathbf{P}^h_{\mathbb{G}}(v(\mathbf{x}_i), I)| \neq 1$. Indeed, any such valuation will be a witness that pd-correctness is violated.

It remains to show that $(i)$ and $(ii)$ can be done in polynomial time. Indeed, $(i)$ is straightforward. For $(ii)$, we can modify the construction in Lemma 17 so that is computes the intersection of the hash destinations as well (instead of only checking for emptiness). ◀

**Proof of Theorem 9.** It is easy to observe that if any relation in a query is non-redundant (which we can check using Algorithm 1 by setting $\mathbf{A} = \mathsf{att}(R)$), then p-correctness implies pd-correctness. However, this condition is not necessary. To address this issue, we consider a generalization of non-redundancy from relations to atoms. Given a co-hash graph $\mathbb{G}$ and a CQ $q$, we say that an atom $R_i(\mathbf{x}_i)$ in $q$ is *dominant* if the root path of $R_i$ contains no other relation that appears in the body of $q$. To compute whether a dominant atom $R_i(\mathbf{x}_i)$ is non-redundant, we simply modify Algorithm 1 such that the closure in Line 16 includes $A = A'$ whenever two attributes $A, A'$ of $R_i$ have the same variable in $R_i(\mathbf{x}_i)$. Now, let $\mathbb{G}$ be a co-hash graph, and $q$ a full CQ such that $q$ is p-correct under $\mathcal{P}_{\mathbb{G}}$. Then, $q$ is pd-correct under $\mathcal{P}_{\mathbb{G}}$ iff there exists a dominant atom in $q$ that is non-redundant.

One direction of the proof is straightforward. If some dominant atom $R_i(\mathbf{x}_i)$ is non-redundant, then the tuples for any valuation $v$ must meet on the unique location where $v(\mathbf{x}_i)$ is assigned to. For the other direction, suppose that $R_i(\mathbf{x}_i)$ is a dominant and redundant atom. Let $D[q]$ be the canonical instance for $q$ with labelled nulls (and constants). Let

$t \in D[q]$ be the tuple corresponding to the atom $R_i(\mathbf{x}_i)$. Following the proof in Lemma 33, we can construct an instance $I = P_1 \cup P_2$ such that $t \in I$, and $P_1, P_2$ are two terminating paths for $t$ in $I$ with hash destinations $h_1$ and $h_2$ respectively, where $h_1 \neq h_2$. Consider now two instances: $I_1 = D[q] \cup P_1$ and $I_2 = D[q] \cup P_2$. Since there is no other relation of $q$ in the root path of $R_i$, the tuple $t$ must have $h_1, h_2$ as its only hash destinations in $I_1, I_2$ respectively. Moreover, since $q$ is p-correct, it must be that all tuples in $D[q]$ have the same hash destination $h_1$ in $I_1$, and $h_2$ in $I_2$.

Finally, consider the instance $I' = D[q] \cup P_1 \cup P_2$. We can now argue that all tuples of $D[q]$ will end up in both $h_1, h_2$, thus proving that the query is not pd-correct. Indeed, suppose for the sake of contradiction that they end up in $h_1$ but not $h_2$. Then, there exists a tuple $s \in D[q]$ that is assigned to $h_2$ in $I_2$, but not in $I'$. For this to have happened, it must be the case that some tuple $s' \in D[q]$ has an empty upwards join in $I_2$, but not in $I'$, where it joins with some tuple from $P_1$. But then, the hash signature of $s'$ is equal to $h_2$. Since $P_1, P_2$ differ only in their non-join values, this implies that $s'$ would be equal to $h_1$ as well, so $h_1 = h_2$, a contradiction. Since we can check non-redundancy in polynomial time, the above algorithm implies a polynomial time algorithm for pd-correctness. ◀