Scalable Social Coordination with Group Constraints using Enmeshed Queries

Jianjun Chen[*]
Google Inc.
Mountain View, CA, USA
jianjunc@google.com

Ashwin Machanavajjhala[*]
Duke University
Durham, NC, USA
ashwin@cs.duke.edu

George Varghese[*]
Microsoft Research
Mountain View, CA, USA
varghese@microsoft.com

ABSTRACT

While specific forms of social coordination appear in tools such as Meetup and in game platforms such as XBox LIVE, we introduce a more general model using what we call *enmeshed queries*. An enmeshed query allows users to declaratively specify an intent to coordinate with other users (who they may not know a priori) by specifying constraints on who/what/when as well as on the composition of the group, such as the desired group size. The database returns a group of users who have registered queries with matching intents. Enmeshed queries are continuous, but new queries (and not data) answer older queries; the group constraints and the ability to coordinate with unknown partners make enmeshed queries differ from entangled queries, publish-subscribe systems, dating services and nested transactions. While even offline group coordination using enmeshed queries is NP-hard, we introduce efficient heuristic algorithms that can scale to millions of queries, and find 86% of the matches found by an optimal algorithm using 40 microseconds per query on a 2.5 GHz server machine. We conclude by describing potential generalizations that add prices, recommendations, and data mining to basic enmeshed queries.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms

Keywords

social coordination, enmeshed queries

1. INTRODUCTION

While social networks like Facebook are widely used for interacting with friends, they seem less useful for planning and *coordinating* with other people. But, this is a significant part of what being *social* entails — we go to birthday parties, we play sports, we fraternize with like-minded people on even obscure topics such as fly-fishing. Further, the default method of coordinating with people by phone/email is tedious, and sometimes hard to drive to closure. Thus social coordination seems the next step beyond social networking.

There are a number of ad hoc social coordination tools already in the market. These include `evite.com` (coordinator specifies list of invitees and an event), `doodle.com` (users specify a range of times for a meeting and the coordinator can look for common times), `meetup.com` (users specify a zipcode and a topic, and then browse through the groups returned by the system), `foursquare.com` (users check into a database of locations, and can query the system for the location of their friends), and `groupon.com` (users specify a location and interest, and then browse through a set of deals; however, deals only take effect when a critical mass of users sign up for the deal).

Existing social coordination tools have four disadvantages:

- *Ad Hoc:* Each system is tailor-made to a specific application although they have many abstract features in common.
- *Limited Query Capability:* While most provide a rudimentary query facility, they also involve considerable browsing and selection by the user.
- *Not Continuous:* If there is no current choice that satisfies the user, the user must retry later: synchronous polling is needed as opposed to asynchronous notification by the system.
- *Lack of fine-grained group size control:* Users cannot specify limits on the group size.

In contrast, our goal is to develop tools for *fluid* social coordination which have the following properties.

- *Generic Platform:* We abstract coordination as finding matches on key attributes such as people, activity, location, and time in a common platform that many social applications can use.
- *Declarative Queries:* Users specify predicates that prescribe the kind of groups they wish: the system matches users to groups without need for browsing.
- *Continuous:* If there is no current choice, the system will retain the user query and subsequently attempt to match this query when future queries enter.
- *Group Constraints:* Users can specify group constraints (such as bounds on group size).

[*]Work done while the authors were at Yahoo! Research.

By fluid, we mean that coordination is not limited to static groups such as friends; instead, we allow coordinating even with strangers, and with different sets of people for different activities. Granovetter [4] has argued that novel information flow (such as job opportunities) typically occurs through "weak" ties (e.g., casual acquaintances) than through "strong" ties (e.g., friends and family). We suggest that allowing people to perform fluid social coordination may encourage the formation of weak ties.

Fluid coordination appears in a number of real world applications. Consider multiplayer online gaming where users wish to form groups whose sizes can depend on the game. The user is indifferent to player identities except that they be close by (to reduce latency), and have similar Internet access speeds and game ratings. By allowing a user, of say Xbox LIVE, to specify parameters in these three key attributes and a group size (say 4 to play Halo), our system can match users up. As a second example, consider finding 3 partners among all Yahoo! employees for playing doubles tennis in Sunnyvale at 4 pm on the weekend. Private data collection is a third example. Users may allow a hospital to publish their information only if they share the same "quasi-identifying" attribute with a group of $k-1$ other users (k-anonymity [13]), or if additionally ℓ distinct diseases appear in their group (ℓ-diversity [9]).

Complete fluidity may alarm some users, especially when considering physical activities such as tennis, where a user may not wish to play with another player, say, with a criminal record. We allow users effectively to "scope" their coordination intents using constraints. For instance, a user can request to only play with other users affiliated with Duke University (i.e., with a `duke.edu` email suffix). Users can set the scope to be wide enough to match their query and yet narrow enough to stay in their comfort zone.

In this paper, we present a system that can support fluid coordination as described in the above examples. In this system, users describe their coordination intent using a novel concept called *enmeshed queries*, which allow a user to specify coordination constraints, like who/what/when, and constraints on coordinating group, like on the number of people forming a group. The formalization and scalable implementation of these queries is the main contribution of this paper.

From an algorithmic viewpoint, think of the set of enmeshed queries as forming the nodes of a graph. We place an edge from node R to node S if R is *compatible* with S. For example, the queries of two tennis players who have compatible ratings who wish to play at the same time and the same place will be linked by an edge. The query system attempts to carve this graph into cliques such that each clique satisfies its group size constraints; for example, one clique could be a set of 4 compatible tennis players to play doubles.

Contributions and Paper Outline: Our contributions are as follows.

- First, we formalize the problem of fluid social coordination by defining the concept of enmeshed queries.
- Second, we show that optimal algorithms for matching enmeshed queries are NP-hard; however, we introduce heuristics that are close to optimal on synthetic workloads.
- Third, we describe an implementation that can scale to millions of concurrent enmeshed queries.

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 describes enmeshed queries, and theoretical hardness results are described in Section 4. Algorithms are described in Section 5. Section 6 presents an evaluation that suggests that our algorithms can scale to millions of queries per second. Section 7 provides a discussion about user selected coordination and system selected coordination. We conclude (Section 8) with a list of future research directions.

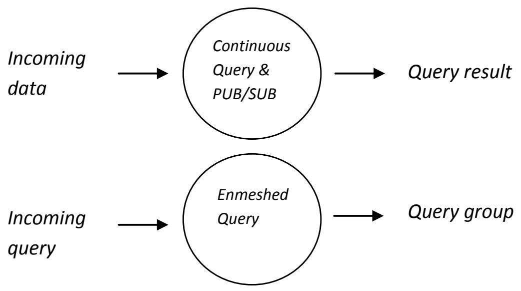


Figure 1: Enmeshed query versus continuous query and pub/sub.

2. RELATED WORK

Enmeshed queries are most closely related to entangled queries [5, 6, 11]. The seminal paper on entangled queries [5] required users to explicitly specify partners they wish to coordinate with (e.g., tennis with John and Sarah). More recent work [11] allows some flexibility in choosing partners by allowing an entangled query to specify *one* partner to be any friend (or someone related to the user through a prespecified binary relation). While enmeshed and entangled queries share the goal of declarative social coordination, there are significant differences. First, enmeshed queries allow group constraints such as bounds on group sizes; entangled queries do not. Second, enmeshed queries are designed for fluid coordination – partners need not be explicitly specified, and indeed may not even be friends. Third, enmeshed queries use online algorithms that match a newly arrived query with a subset of existing queries. Our online algorithms are designed to scale to millions of concurrent queries. In contrast, entangled queries consider (offline) coordination among a given small set of queries (1000s of queries in [5, 11]). Entangled queries, on the other hand, have considerable power in specifying constraints on external relations that may be the final object of coordination (e.g., a tennis reservation at a specified court). An interesting future direction is to combine the power of these two formalisms.

Pub-sub systems [12] and continuous query systems [14, 1] also provide declarative continuous query evaluation, but each query is logically independent; further there are no constraints on groups. On the other hand, enmeshed queries find matches among queries. Their differences are illustrated in Figure 1. Our approach to find candidate query matches is related to work in efficiently evaluating boolean expressions [2], and can utilize index structures proposed for implementing scalable ranked pub/sub [10]. Finally, enmeshed queries differ from nested transactions [8], since at query time the system does not know which other enmeshed

	Group Constraints	Independent Queries	Query Life	# of queries
Enmeshed	Yes	No	Short	Millions
Entangled	No	No	Short	Thousands
Continuous Queries	No	Yes	Long	Varying
Pub-Sub	No	Yes	Long	Millions

Table 1: Comparison among continuous queries, pub/sub systems, entangled queries and enmeshed queries.

queries it is waiting on. Table 1 summarizes these comparisons.

There is also related work on team formation in social networks [7] that studies the following problem: given a set of people (i.e. nodes in a graph) with certain skills and communication costs across people (i.e. edges in the graph), and a task T that requires some set of skills, find a subset of people to perform T with minimal communication costs. They prove that such problems are NP hard, and describe heuristics to reduce computation. Our problem is not the same as task formation because tasks are not explicit first-class entities in enmeshed queries but are, instead, implicit in the desires of users. Further, our metric is maximizing matches and not minimizing communication.

3. MODEL AND PRELIMINARIES

We formally define enmeshed queries and the problem of social coordination.

Users: Consider a set of users $\mathcal{U}$ who wish to participate in coordination tasks. Each user is associated with a set of attributes $\mathcal{A}_{\mathcal{U}}$. Examples of user attributes include age, home location, tennis rating, etc. Denote by $dom(A)$ the domain of an attribute A, and by $dom(\mathcal{A}) = \times_{A\in\mathcal{A}} dom(A)$ the cross product of the domains of a set of attribtues $\mathcal{A}$. A point $\vec{x} = [x_1, \ldots, x_k]$ is a multidimensional value from $dom(\{A_1, \ldots, A_k\})$.

Enmeshed Queries: Users specify their intent to coordinate using one or more enmeshed queries. An enmeshed query q is associated with (i) a unique user $q.user$, and (ii) a set of free coordination variables $q.\vec{x} = [q.x_1, q.x_2, \ldots, q.x_{|C|}]$. Each free variable $q.x_i$ takes values from a distinct coordination attribute $A_i \in \mathcal{A}_C$. Examples of coordination attributes are time(when), location (where), activity type (what), etc. Enmeshed queries define coordination intent by specifying constraints on the coordination variables and user attributes. Formally, an enmeshed query is a triple $q = (\mathcal{S}, \mathcal{J}, \mathcal{G})$, where $\mathcal{S}$ is a set of selection constraints, $\mathcal{J}$ is a set of join constraints, and $\mathcal{G}$ is a set of group constraints. We define $\mathcal{S}$, $\mathcal{J}$, and $\mathcal{G}$ in turn.

First, $\mathcal{S}$ specifies the query's coordination intent $\mathcal{S}$ by specifying sets of possible values for each coordination variable. For instance, a user may want to play tennis or squash, between 4 and 6 PM, in either Cupertino or Mountain View. More formally, coordination intent $\mathcal{S}$ is specified as a conjunction of *selection constraints*, where each selection constraint is of the form $x_i \in S$, where $S \subseteq dom(A_i)$. Note that queries may not pose any constraint on some coordination attributes; this is modeled using $S = dom(A_i)$.

Second, $\mathcal{J}$ specifies additional *join constraints* over the user attribute values on pairs of enmeshed queries. Examples include: Alice would like to coordinate only with her friends, and Alice would like to play tennis with users who have a higher rating. If Alice knew a priori that she was coordinating with Bob, then Alice could easily express the above rating constraint as $Bob \in Alice.Friends$ and $(Bob.rating > Alice.rating)$. However, when declaring coordination intent, Alice does not know who she is coordinating with. To describe join constraints, we introduce the notation **that** to refer to attributes of other users who may potentially coordinate on a task. Thus the above constraints in Alice's query q can be written as $\textbf{that}.id \in user.Friends$ and $\textbf{that}.rating > user.rating$. $\mathcal{J}$ is a conjunction of such individual join constraints.

Finally, $\mathcal{G}$ specifies predicates on allowable groups. For example, one could specify the average rating of the group of individuals for a doubles tennis game. For this paper, we focus on the simplest and most useful group constraint, a *cardinality constraint* such as "want to play tennis with at least 2 or at most 4 individuals". Formally, a cardinality constraint $\mathcal{G}$ is represented as a pair of integers $\{(lb, ub)\}$, where $2 \le lb \le ub$. This ensures a constraint that the query q must co-ordinate with a group of at least lb and at most ub queries (inclusive of q).

Example: A user wanting to play tennis in Cupertino (C) or Sunnyvale (S) at 8 PM with 2,3 or 4 people with an equal or higher rating can be written as the following query $q = (\mathcal{S}, \mathcal{J}, \mathcal{G})$:

$$\mathcal{S} : \{A_{time} \in \{8pm\}, A_{loc} \in \{C, S\}, A_{sport} \in \{Tennis\}\}$$
$$\mathcal{J} : \{\textbf{that}.A_{rating} \ge A_{rating}\}$$
$$\mathcal{G} : \{[2, 4]\}$$

The query q can alternately be represented as a conjunction of the selection, join and group constraints as follows:

$$A_{time} \in \{8pm\} \wedge A_{loc} \in \{C, S\} \wedge A_{sport} \in \{Tennis\}$$
$$\textbf{that}.A_{rating} \ge A_{rating} \wedge \text{card}(q) \in [2, 4]$$

where $\text{card}(q) \in [2, 4]$ is a shorthand representation of the group cardinality constraint.

Matching Queries: We define the semantics for how/when coordination is achieved in stages by adding in each type of constraint in turn. First, we say that a set of queries $q_1, \ldots, q_k$ are *jointly satisfied* if there is a point $\vec{p} \in dom(\mathcal{A}_C)$ such that for every query q_i and selection constraint $x_j \in S_{ij}$ in q_i, we have $p_j \in S_{ij}$. Next, we incorporate join constraints as follows. Two queries q_1 and q_2 are said to *match* if (i) their selection constraints are jointly satisfied, and (ii) join constraints on q_1 and q_2 are satisfied. Finally, we add group cardinality constraints:

DEFINITION 1 (COMMITTABLE QUERY GROUP). *A group of queries $Q \subseteq \mathcal{Q}$ is called a* committable query group *if*

- *Selection conditions on queries in Q are jointly satisfied,*
- *$\forall q_1, q_2 \in Q$, q_1 and q_2 match on join constraints, and,*
- *$\forall q \in Q$, $lb_q \le |Q| \le ub_q$, where $\mathcal{G} : [lb_q, ub_q]$ is the group cardinality constraint for q.*

A committable group can be returned by the system as a valid set of users who can be matched together. All queries in such a group are called *committed.*

Problem Statement: User coordination is now reduced to the problem of finding committable query groups. Note that the problem is online – the system cannot wait till all the queries have been submitted.

Definition 2 (Group Coordination Problem). *Given a stream of enmeshed queries $\mathcal{Q} = q_1, q_2, \ldots, q_n$, find sets of committable query groups that maximize the number of committed queries. We denote by* OPT$(\mathcal{Q})$ *the* offline optimal*, or the maximum number of committable queries that can be committed if all the queries are known upfront. Our goal is to design an algorithm* ALG *such that for any finite subsequence Q of $\mathcal{Q}$, the number of queries committed (*ALG(Q)*) is as close to* OPT(Q) *as possible.*

We consider the following optimization metrics:

- *Throughput:* This can be measured in two ways – committed query cardinality, and committed group cardinality. *Committed query cardinality* is measured as the total number of queries that are committed. *Committed group cardinality* measures the total number of groups that are committed. Both metrics can reflect revenue if the system receives a reward (e.g., ad revenue) for every group or query that it commits. Clearly, a system that optimizes group cardinality may prefer smaller groups over larger ones. We use query cardinality in our evaluation.
- *Processing Time:* The average time taken to process a query. This reflects the costs of the system providers.
- *Latency:* The duration between when an enmeshed query first entered the system and when it was actually committed as a group. Duration is measured in terms of the number of queries elapsed between when the query entered and when it was committed. This is a measure of user satisfaction.
- *Fairness:* We wish to penalize algorithms that are biased towards returning smaller group rather than larger ones. A simple measure we use to compare algorithms for this bias is the average size of committed groups.

While maximizing the throughput is our main goal, we will also experimentally measure query processing time, latency and fairness with respect to group size of our algorithms.

4. COMPLEXITY

We show complexity results for the Offline and Online group coordination problems.

4.1 Offline Problem

In this section we show that finding the offline optimal $OPT(\mathcal{Q})$ for the group coordination problem is NP-hard, whether we want to maximize the number of committed queries or groups. When maximizing queries, or groups, we show the problem is NP-hard even when there is a single coordination attribute.

Lemma 1. *Given a set of enmeshed queries $\mathcal{Q}$ with a single coordination attribute A, the problems of computing the maximum number of committed queries and committed groups are NP-hard.*

Proof. (sketch) We show hardness via a reduction to the maximum 3-dimensional matching problem, a classic NP-complete problem [3]. An instance of the 3-D matching problem consists of disjoint sets X, Y, Z and a subset $T \subseteq X \times Y \times Z$ of triples. $M \subseteq T$ is called a *matching* if for any two distinct triples $(x, y, z), (x', y', z') \in M$, we have $x \neq x', y \neq y'$ and $z \neq z'$.

Given an instance of 3-D matching, we construct an instance of the group commit problem as follows. There is a single coordination attribute A whose domain is $X \times Y \times Z$. Without loss of generality, we can assume that every element in $v \in X \cup Y \cup Z$ appears in at least one triple in T (otherwise 0-degree elements can be removed from the problem). With every element v, we associate a query q_v having a cardinality constraint of 3, and a coordination $A \in S$, where S is the set of triples that contain v.

Every committable group corresponds to a unique triple (x, y, z). Moreover, since no two committed groups can share a query, the set of committed groups gives a 3-D matching. Thus the size of the maximum 3-D matching corresponds to maximum number of committable groups. Finally, the number of committed queries is just 3 times the number of committed groups, and thus finding the maximum number of committed queries allows one to compute the maximum 3-D matching. Hence, both problems of computing the maximum number of committed queries and groups is NP-hard. $\square$

4.2 Best-Effort Online Coordination

In this section, we consider a special class of online algorithms. An algorithm is *best-effort* if when a query q enters the system at most one committable group is returned and any returned committable group includes q. It may not return a committable group even when one exists. By contrast, an algorithm is *optimal best-effort* if when a query q enters the system exactly one committable group is returned that includes q, if such a committable group exists.

Best-Effort algorithms are attractive because they ensure constant progress. Second, best-effort algorithms are k-competitive, where k is the maximum cardinality constraint of a query. That is, the number of queries committed by a best-effort algorithm is at least $1/k$ times the number of queries committed by an offline optimal algorithm. The competitive result follows because for any group output by the best-effort algorithm, in the worst case, the offline optimal might have used each query in the group to commit a separate group. However, *optimal* best-effort algorithms are hard to design.

Lemma 2. *Best-effort algorithms are k-competitive.*

Proof. The following construction shows that the competitive ratio is at least k. Consider k^2 queries which are numbered $q_1, \ldots, q_{k^2}$ in the order they enter the system. Queries $q_1, \ldots, q_k$ form a committable group. Additionally, q_i and $q_{i(k-1)+2}, \cdots, q_{(i+1)(k-1)+1}$ form k different committable groups for $i = 1$ to k. Figure 2 shows the construction for $k = 4$; nodes are queries, edges represent whether or not queries match. All queries have cardinality constraint $= 4$. More specifically, for queries 1-4, q_i's constraint is $A \in \{x, i\}$. For queries connected to q_i $(i = 1, 2, 3, 4)$, the constraint is $A \in \{i\}$.

An optimal solution is to commit $(1, 5, 6, 7)$, $(2, 8, 9, 10)$, $(3, 11, 12, 13)$ and $(4, 14, 15, 16)$ (the k groups in the general case). However, a best effort algorithm would greedily commit $(1, 2, 3, 4)$, $(q_1, \ldots, q_k$ in the general case). Thereafter no other query can commit. Thus the competitive ratio is at least k.

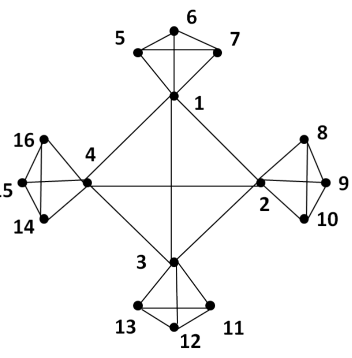


Figure 2: Example illustrating the competitive ratio of best-effort algorithms. Nodes are queries, edges represent whether or not queries match. All queries have cardinality constraint = 4. Numbers also represent the order in which queries enter the system.

For any group output by the best-effort algorithm, in the worst case, the offline optimal might have used each query in the group to commit a separate group. Since the max cardinality constraint is k, the competitive ratio is $\leq k$. □

However, we next show that best-effort algorithm also are hard to design. When a new query comes in, best-effort algorithms need to find a committable group if there is one. But, this problem is NP-hard in the presence of join constraints. In the absence of join constraints, we present a PTIME best-effort algorithm which will be the scaffolding for our efficient algorithms presented in the next section.

LEMMA 3. *Computing an optimal best-effort solution is NP-hard, even if the number of attributes is a constant. Suppose the number of attributes is a constant. Ensuring that an algorithm always finds a committable group if there is a committable group is NP-hard.*

PROOF. (sketch) Let $\mathcal{S}$ be the set of query groups that can be committed due to adding q to $\mathcal{Q}$. Suppose there do not already exist any committable groups in $\mathcal{Q}$. Hence, for all groups $S \in \mathcal{S}$, $q \in S$.

It is easy to verify that the problem is in NP. Given a group, we can efficiently check whether it is committable.

To show it is hard we present a reduction from the clique decision problem: the problem of determining whether there is a clique of size k in a graph $G = (V, E)$ is well known to be NP-hard. Given an instance of the clique problem, we construct an instance of our problem with a single attribute A. For every edge $e = (x, y) \in E$, we have a value $a_e \in A$. We use e to represent both (x, y) or (y, x).

We construct one user u_x who initiates query q_x for every $x \in V$. Each query has cardinality equal to $(k + 1)$. *The queries are such that* (q_x, q_y) *matches if and only if* $(x, y) \in E$. This can be achieved just by constructing two user attributes A_{ID} and A_{friend}. For each user u_x, $A_{ID} = x$ and $A_{friend} = \{y|(x, y) \in E\}$. Finally, query q_x has a join constraint, $\textbf{that}.A_{ID} \in A_{friend}$. This requires that each query q_x be satisfied only by a q_y, where $(x, y) \in E$.

We have a new user u who initiates a query q that matches all queries we constructed (by not having any constraint for q) and also having a cardinality $k + 1$. If we find a committable group in the new instance, we get a clique in the original graph. Thus the problem is NP-hard. □

Algorithm 1 BMA: Basic Matching Algorithm

```
1: Input: A set of enmeshed queries $Q$, a new query $q$, an inverted index PINDEX
2: Output: A committable group $C$ (and $Q \leftarrow Q - C$),
3:           or null (and $Q \leftarrow Q \cup \{q\}$).
4: Let $P \subseteq dom(\mathcal{A}_\mathcal{U} \cup \mathcal{A}_C)$, s.t., $\forall \vec{p} \in P$, $\vec{p}$ satisfies $q$
5: for $\vec{p} \in P$ do
6:    // STEP 1: Identify queries whose selection constraints are satisfied by $\vec{p}$.
7:    $PQS_p \leftarrow$ PINDEX.lookup($\vec{p}$)
8:    $CQS_p \leftarrow$ subset of queries in $PQS_p$ satisfying $\vec{p}$
9:
10:   // STEP 2: Identify subset of queries whose join and cardinality constraints are satisfied
11:   Set $S_p \leftarrow \{q\}$
12:   for $q' \in CQS_p$ // in random order do
13:      if ($q'$ matches $\forall q \in S_p$) then
14:         $S_p \leftarrow S_p \cup \{q'\}$
15:         $C \leftarrow$ FINDCOMMITTABLEGROUP($S_p$)
16:         if $C$ is not null then
17:            Remove all queries in $C$ from $Q$ and PINDEX.
18:            Return $C$
19:         end if
20:      end if
21:   end for
22: end for
23: // No committable group found
24: Add $q$ to QueryTable and to PINDEX.
```

5. SCALABLE GROUP COORDINATION

Given that optimal best-effort algorithms are NP-hard, we present four heuristic algorithms that will work well on workloads satisfying the following assumptions:

A1. High Selectivity: We assume most queries are selective, and hence, can be indexed based on some coordination attributes. First, people know what they want: queries like "play any sports at any time and anywhere" are not typical. Second, selectivity can be enforced by application user interfaces that disallow wildcards, and enforce limited disjunction (e.g., at most two locations allowed in a query).

A2. Small Group Coordination: We also assume that the average cardinality of queries are small, e.g. < 50. This ensures that queries do not wait too long to be committed.

All our algorithms use the following structure. They all start by using a subset of the most selective attributes in a query as an index to quickly find a (hopefully small) "short list" of potentially matching queries. For example, out of a million concurrent enmeshed queries, only 50 queries in the system may specify 'playing Tennis in Cupertino at 3 PM next Tuesday'. While the first pruning step only considers selection constraints, the second step traverses the "short list" to greedily attempt to build a committable group while also incorporating join and cardinality constraints.

5.1 Basic Matching Algorithm (BMA)

Basic matching algorithm, BMA (Algorithm 1), is a best-effort algorithm that traverses the "short list" in random order, and stops when it finds the first committable group.

Given a query q, BMA iterates over all possible points $\vec{p} \in \mathcal{A}_C$ that satisfy the selection constraints of q in random order. For each $\vec{p}$, it first finds a set of queries (PQS_p) that partially match q using PINDEX, which is an in-memory inverted hash index defined over a subset of coordination attributes (e.g., location and time). This index (built using techniques in [2]) can support hierarchical values, e.g. weekday instead of dates. Since PINDEX is only on a subset of attributes, not all queries in PQS_p are jointly satisfied by $\vec{p}$. Hence, next a subset of queries satisfying $\vec{p}$, called CQS_p, is computed. CQS_p is the "short list" we referred to earlier. If assumption *A1* holds, $|CQS_p|$ will be small.

However, not all pairs of queries in CQS_p may match according to the join constraints. Hence, BMA next greedily computes $S_p \subseteq CQS_p$, a subset of queries that match (and satisfy $\vec{p}$) as follows: Starting with $S_p = \{q\}$, BMA iteratively adds a randomly chosen query $q' \in CQS_p$ to S_p if it matches every other query in S_p (based on join constraints). As the following example illustrates, there may be many choices for S_p for the same CQS_p.

EXAMPLE 1. *Consider queries with a user attribute* $A_{rating} \in \{1, 2, \ldots, 5\}$. *Let* q *be a query with no join constraints, with* $A_{rating} = 3$. *Suppose* CQS_p *consists of* q_1, q_2 *and* q_3, *such that* $q_1.A_{rating} = 4$, $q_2.A_{rating} = q_3.A_{rating} = 2$, *and* $card(q_2) = card(q_3) = 3$. *Suppose* q_1 *has a join constraint* $\mathbf{that}.A_{rating} < A_{rating}$ *(only play with lower ranked players), and* q_2, q_3 *have join constraints* $\mathbf{that}.A_{rating} = A_{rating} + 1$ *(only play with players ranked 3). Then if the queries are considered in the order* q_1, q_2, q_3, *the resulting* $S_p = \{q, q_1\}$. *If* q_2 *or* q_3 *is considered before* q_1, *then the resulting* $S_p = \{q, q_2, q_3\}$.

Every time a new query is added to S_p, the FINDCOMMITTABLEGROUP subroutine attempts to find a committable group that satisfies cardinality constraints as follows: Every query added to S_p is also added to an inverted index from possible group sizes to queries, called CINDEX. A query with multiple group sizes appears multiple times in CINDEX, one for each group size. BMA iterates over CINDEX to check whether there is some m such that there are $\geq m$ queries in S_p that permit a group of size m. BMA picks some group of m queries. Assumption *A2* ensures that the average number of iterations over CINDEX is small. If a group is formed, all queries from the group are removed from the system. Otherwise, we add current query q into the CINDEX.

EXAMPLE 2. *For instance, suppose* $\{q_1, q_2, q_3, q_4, q_5\}$ *is a set of matching queries in* S_p *with cardinality constraints* $2, [2, 3], [2, 3], [2, 3]$ *and* 3 *respectively. Then* CINDEX *will contain* 2 *entries:* $2 \to \{q_1, q_2, q_3, q_4\}$ *and* $3 \to \{q_2, q_3, q_4, q_5\}$. *Any pair from* $\{q_1, q_2, q_3, q_4\}$ *or any subset of size 3 from* $\{q_2, q_3, q_4, q_5\}$ *is committable.* FINDCOMMITTABLEGROUP *will return some size 3 subset of* $\{q_2, q_3, q_4, q_5\}$.

Note that in the absence of join constraints, BMA returns an optimal best-effort solution. For each point $\vec{p}$, CQS_p precisely corresponds to all the queries that are jointly satisfied by $\vec{p}$, and CINDEX correctly identifies a commitable, if there exists one. Moreover, if $P \subseteq \mathcal{A}_C$ is the set of points that jointly satisfy the new query q, then the algorithm takes $\sum_{\vec{p} \in P} |PQS_p|$ time to terminate. While in the worst case, this can be as large as $c \cdot (|Q| \cdot |P|)$, for some constant c, the run time is much smaller when we have a selective PINDEX.

5.2 Extensions to BMA

BMA randomly traverses the "short list" and returns the first committable group it finds. The next three algorithms extend BMA by more carefully choosing a committable group (among many alternatives), delaying committing to improve throughput, and by using more intelligent traversal orders.

5.2.1 NES: No Early Stop

BMA may choose to commit a group even before all the queries in CQS_p are seen. NES, on the other hand, computes S_p by considering *every* query in CQS, and then calls FINDCOMMITTABLEGROUP to find the largest subset of S_p satisfying cardinality constraints. In Example 2, suppose the queries in S_p are considered in order of their indices. BMA will commit only 2 queries, (q_1, q_2) after seeing the first two queries. On the other hand, NES waits to see all queries and commits (q_2, q_3, q_4).

5.2.2 DELAY: Delayed Matching

Both BMA and NES are best-effort – when a new query arrives, if a group can be committed, it will be committed. However, delaying the matching of smaller groups may help find a larger group later.

EXAMPLE 3. *Consider again Example 2. Suppose queries come in the order of their index. After* q_1 *and* q_2 *enter the system, any best-effort algorithm would commit the group* (q_1, q_2). *Similarly, after* q_3 *and* q_4 *enter the system, any best-effort algorithm would commit the group* (q_3, q_4). *However, if we waited till* q_5 *came into the system, NES can commit* (q_3, q_4, q_5), *thus committing all queries.*

We implement DELAY as follows. Let the *age* of a query denote the time elapsed since it was inserted into the system[1]. When a query first arrives at time t, we compute its CQS_p, and proceed to find a committable group only if the average age of the CQS_p is greater than a threshold τ. Otherwise, q is inserted into a scheduler, akin to a timing wheel, to be reevaluated at time $t + \tau$. A query is reevaluated at most once. DELAY can increase processing time and query latency, as some queries are evaluated twice. If at most m queries simultaneously enter the system, DELAY returns no more than $2m$ committable groups: $\leq m$ groups from new queries and $\leq m$ from delayed queries.

5.2.3 Query Matchibility

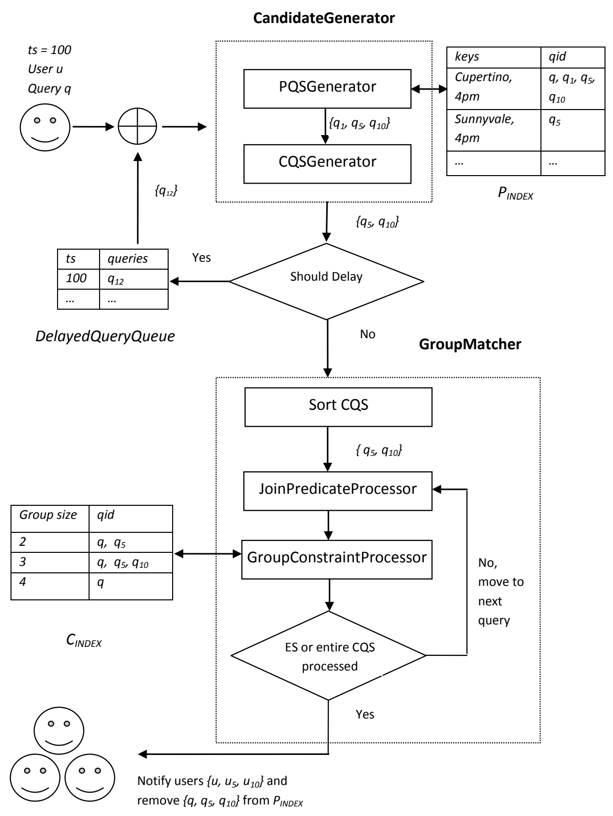
All of the previous algorithms may choose to commit a group that is not optimal in terms of future queries.

EXAMPLE 4. *Consider queries with an attribute* A_{where} *with domain* $\{Cupertino, Sunnyvale, Campbell\}$. *Suppose* $A_{where} = Campbell$ *is very rare. Consider a new query* q *with* $A_{where} = Campbell$ *and cardinality constraint* $[2, 3]$. *Suppose its* CQS *contains three queries* q_1, q_2, q_3 *with* q_1, q_2 *having* $A_{where} \in \{Cupertino, Campbell\}$ *and* q_3 *requiring* $A_{where} \in \{Campbell\}$, *resp. Moreover,* q_1, q_2 *have cardinality constraint* 3, *while* q_3 *requires* 2. *Then, returning the group* (q, q_3) *is better than returning* (q, q_1, q_2) *(even though the latter is larger), since it is more likely that a Cupertino query will arrive in the future than a Campbell query.*

We call the propensity of a query to be matched as its *matchibility*, and is intuitively the expected number of other queries that will match q. Intuitively, a query has low matchibility either because its selection or join constraints are hard to satisfy, or because it has large cardinality. We use a simple online approach to estimate matchibility.

We define the matchibility of a query q as the number of times q appears in a CQS_p before it is committed. Intuitively, a query that appears more frequently in the CQS_p

[1]We assume each query is inserted at a new time instant.



*$u_1.A_{rating}$ = 2000, q_1: A_{loc} ε {Cupertino}, A_{time} ε {4pm}, A_{sport} ε {Chess}, **that**.A_{rating} = 2000, card(q_1) ε [2,4]*
*$u_5.A_{rating}$ = 3, q_5: A_{loc} ε {Cupertino,Sunnyvale}, A_{time} ε {4pm}, A_{sport} ε {Tennis}, **that**.A_{rating} = 3, card(q_5) ε [2,3]*
*$u_{10}.A_{rating}$ = 3, q_{10}: A_{loc} ε {Cupertino}, A_{time} ε {4pm,5pm}, A_{sport} ε {Tennis}, **that**.A_{rating} = 3, card(q_1) ε [3,3]*
*$u.A_{rating}$ = 3, q: A_{loc} ε {Cupertino}, A_{time} ε {4pm}, A_{sport} ε {Tennis}, **that**.A_{rating} = 3, card(q_1) ε [2,4]*

Figure 3: Enmeshed query system and data flow.

of other queries should have a higher chance to find matches. The initial value of a query's matchibility is

$$m(q) = n_q + c/lb_q$$

where c is a constant, lb_q is the lower bound on the cardinality constraint in the query, and $n_q = ub_q - lb_q + 1$ is the number of possible cardinality values that satisfy q. For example, if a query has a cardinality constraint $[4, 6]$, then its initial matchibility is set to $c/4 + 3$. The motivation is that smaller groups are more matchable; further, a wide range of cardinalities is more matchable. Every time q appears in some other query's CQS_p, $m(q)$ is incremented by 1.

We consider two simple heuristics – *high matchibility first (HMF)* and *low matchibility first (LMF)* that traverse the "short list" CQS_p in matchibility order. Intuitively, LMF may match more queries than HMF by reserving popular queries for later consideration.

Discussion: The 3 extensions presented in this section are orthogonal, and an algorithm can be implemented with some or all of them together. However, given that DELAY postpones queries for later consideration, it seems unreasonable to not consider all queries. Hence in our implementation, DELAY always implies NES, but not vice versa.

5.3 System Description

Figure 3 depicts key modules and data structures in our enmeshed query system. Our system consists of two main modules: CandidateGenerator, which identifies a small set of candidate matching queries for an incoming query q, and GroupMatcher, which identifies a committable group among the candidates.

As described earlier, CandidateGenerator uses PINDEX to first compute the PQS for q. Recall that PINDEX is an index over a subset of coordination attributes; thus PQS produces a set of queries that match a subset of q's selection constraints. Iterating over PQS to weed out all queries that do not match the remaining selection constraints produces the CQS. Before GroupMatcher takes over, delayed evaluation (DELAY) is implemented as follows: if the average age of CQS is less than a predefined threshold, q is inserted into a DelayedQueryQueue for later reevaluation. If this condition is false or DELAY is not implemented, CQS is passed to GroupMatcher.

GroupMatcher first orders queries in CQS based on the chosen heuristic (e.g. random in BMA, highest matchability in HMF). GroupMatcher then initializes a result set with q. Next, GroupMatcher iterates over each query p in CQS. If p satisfies the join predicate with all queries in the current result set, p is added to the result set and also to CINDEX. Recall that CINDEX an inverted index from possible group sizes to queries in the result set. Unlike PINDEX, CINDEX is a temporary data structure that is deleted after group matching is processed.

If NES is used, all queries in CQS will be considered before GroupMatcher attempts to find a group. If NES is not used, when a query from CQS is added into the result set, GroupMatcher traverses CINDEX and returns "early" if a matching group is found. GroupMatcher traverses CINDEX in decreasing order of cardinality so that whenever a group is found, it is as large as possible. After a group is found, all queries from the group are removed from PINDEX and notifications are sent to users whose queries matched.

Example Execution: Figure 3 describes an example execution of the system. User may want to coordinate on one of two sports – tennis or chess. Hence, each user has two attributes tennis rating $A_{ratingT}$ and chess rating $A_{ratingC}$. There are 3 coordination variables A_{loc}, A_{time} and A_{sports}. We consider the system at timestamp 100, when there are already 3 uncommitted queries in the system q_1, q_5, q_{10} initiated by users u_1, u_5 and u_{10}. A new user u inserts a query q into the system. The queries are described below:

$$q_1 : \begin{array}{c} A_{time} \in \{4pm\} \wedge A_{sports} \in \{Chess\} \wedge A_{loc} \in \{C\} \wedge \\ \textbf{that}.A_{ratingC} = 2000 \wedge card(q_1) \in [2,4] \end{array}$$

$$q_5 : \begin{array}{c} A_{time} \in \{4pm\} \wedge A_{sports} \in \{Tennis\} \wedge \\ A_{loc} \in \{C, S\} \wedge \textbf{that}.A_{ratingT} = 3 \wedge card(q_5) \in [2,3] \end{array}$$

$$q_{10} : \begin{array}{c} A_{time} \in \{4pm, 5pm\} \wedge A_{sports} \in \{Tennis\} \wedge \\ A_{loc} \in \{C\} \wedge \textbf{that}.A_{ratingT} = 3 \wedge card(q_{10}) \in [3,3] \end{array}$$

$$q : \begin{array}{c} A_{time} \in \{4pm\} \wedge A_{sports} \in \{Tennis\} \wedge \\ A_{loc} \in \{C, S\} \wedge \textbf{that}.A_{ratingT} = 3 \wedge card(q) \in [2,4] \end{array}$$

where C and S are short for Cupertino and Sunnyvale, respectively. User u_1 has a chess rating of 2000, while u_5, u_{10} and u have a tennis rating of 3.

Assume that the PINDEX uses A_{loc} and A_{time}. Thus the PQS for q will include q_1, q_5, and q_{10} since they match q's location and time. However, the CQS for q only contains q_5 and q_{10}, since q_1 specifies $A_{sport} \in \{Chess\}$. Assume that

Parameter Name	Domain	Multi-valued	Distribution
Home Location	$\{1, \ldots 10\}^2$	1	Zipfian
Sport	$\{1, \ldots 10\}$	≤ 3	Uniform
Rating	$\{1, \ldots 5\}$	1	Zipfian

Table 2: Parameter distribution for generating users in our synthetic workload.

the average age of the CQS is high enough so that the CQS is passed to the GroupMatcher.

Suppose the queries in the CQS are ordered as $[q_5, q_{10}]$ (say by matchability). Both q_5 and q_{10} pass the JoinPredicateProcessor, as all three queries q, q_5, q_{10} have the same rating which satisfies the join constraint $\textbf{that}.A_{ratingT} = 3$. Based on the ordering, q_5 is first added into the result set and also to CINDEX. Note that q_5 is inserted twice in CINDEX since its group size constraint is [2,3].

If ES is the chosen algorithm, $\{q, q_5\}$ is returned as a committable group as soon as q_5 is processed. On the other hand, if NES is used, no groups are returned until the entire CQS is processed. In this case, q_{10} is inserted into CINDEX with cardinality 3. GroupMatcher then considers the cardinalities in CINDEX in decreasing order: there is only one query q that permits cardinality 4, but there are three matching queries $\{q, q_5, q_{10}\}$ permit cardinality 3. Thus $\{q, q_5, q_{10}\}$ is returned.

Once the system finished processing q, it checks the DelayedQueryQueue for any query that is scheduled for processing at that timestamp. In this example, q_{12} is scheduled to be processes and hence it is reevaluated in similar fashion to q with one difference. DELAY processes each query at most two times. Therefore, q_{12} will be processed irrespective of the average age of its CQS.

6. EVALUATION

We describe our evaluation in terms of an example sports coordination application. In this application, users want to find sports partners that live nearby and with similar ratings. There are three user attributes – home location, sports a user plays and corresponding sport ratings. There are four coordination attributes – location, time, sport and opponent rating. Each user specifies selection constraints on the desired location(s), desired time(s), and sport. Additionally, a user may specify what the opponent's rating must be – this can be captured using a join constraint. For instance, a user wanting to play tennis in Cupertino or Sunnyvale at 8 PM with 2,3 or 4 people with a rating 5 can be written as the following query:

$$A_{time} \in [8PM] \wedge A_{location} \in \{Cupertino, Sunnyvale\} \wedge$$
$$A_{sport} \in \{Tennis\} \wedge \textbf{that}.A_{rating} = 5 \wedge card(q) \in [2, 4]$$

6.1 Metrics and Setup

We implemented all the algorithms presented in the paper, namely BMA, NES, DELAY, LMF and HMF. We compare our group coordination algorithms using system measures, throughput and query processing time, as well as user measures, latency and fairness. These metrics are defined in Section 3. We use a 2 x Xeon L5420 2.50GHz machine running 64bit RHEL Server 5.6 with 16 GB memory.

6.2 Synthetic Workload

Parameter Name	Domain	Multi-valued	Distribution
Location	$\{1, \ldots 10\}^2$	≤ 2	Zipfian
Time	28 days $\times$ 12 hrs	≤ 2	Bimodal
Sport	$\{1, \ldots 10\}$	1	Uniform
Rating	$\{1, \ldots 5\}$	≤ 2	Zipfian
Group Size	$\{2, \ldots, 12\}$	1	Zipfian

Table 3: Parameter distribution for generating queries in our synthetic workload.

Our data trace contains 1 million queries for 200,000 unique users. Each user has have about 5 queries in the workload since each user has a equal chance to be chosen for a query. We also enforce that no more than one query from the same user can share a time slot. Each query in the workload contains a monotonically increasing time stamp, desired location(s), desired time(s), action, desired ratings for potential matching candidates and a range of desired group size. We now specify details for each parameter. The parameters are also summarized in Tables 2 and 3. All Zipf distributions use an exponent parameter of 1 where the second most common frequency occurs 1/2 as much as the first, etc.

Location: Our workload generator generates 100 locations, represented by a two dimensional array location[10][10]. Each user is assigned a location (i.e. location[k][j]) as his/her home location based on a Zipfian distribution that models the fact that some locations (e.g., San Francisco) have more users than others (say Brisbane, CA). Besides the home location, a user query can also choose from among 4 neighboring locations: if a user is at location $[k], [j]$, the neighboring locations are $[k+1][j]$, $[k-1][j]$, $[k][j+1]$, $[k][j-1]$. This models the fact users that only want to play sports in locations "near" their home locations. We allow at most 2 locations specified in a query using a Zipfian distribution wherein it is more probable to generate queries with 1 location (always home location) than 2 alternative locations (home location + 1 choice made uniformly among the 4 neighbors).

Time: Times are represented as hourly slots, e.g. 2/1/2012 3pm. The domain is between 8am-8pm for the following 4 weeks. We assign higher probability when generating slots on weekends than those on weekdays. Times in a query are chosen uniformly from slots in the domain based on their relative probabilities. No more than 2 time slots can be specified in a query; as in the case of location, we use a Zipfian to choose the number of time slots specified in a query, with a higher probability of a query being generated with 1 time slot than with 2 time slots.

Actions and Ratings: Each user has a set plays up to three sports with the exact number being chosen using a Zipfian distribution, with one being the most probable. Once the number of sports is chosen, the specific sports played by a user is chosen uniformly at random from a set of 10 sports. We also assign a rating (r) for each chosen sport to represent a user's skill level, again using a Zipfian distribution. Similar to the case of location, we limit each query to have no more than 2 desired ratings, where most queries will have 1 rating (equal to r), some have two ratings (r and either $r + 1$ or $r - 1$) where the frequency of 1 versus 2 is chosen by a Zipfian. While a user can play at most 3 sports, each user query picks a *single* sport uniformly at random from among the set of sports the user plays.

Group sizes: Group size generation is described in the next section.

6.3 Finding a yardstick for optimality

Given that finding an optimal solution to enmeshed queries is NP hard, a major challenge is measuring how close our matching algorithms come to the optimal solution. In addition, since the total space of attributes (e.g., the cross-product of location, time etc) is large, if we get a low percentage of queries matched by our algorithms, we cannot determine whether the low match percentage is caused by ineffective algorithms or because the workload generated sparse points in a large domain space. To solve this dilemma we generate data traces that only contain queries belong to pre-matched groups. The advantage of this approach is that we know an optimal solution is able to find matches for all queries in the workload, which then serves as a yardstick for our heuristic algorithms.

More precisely, we first generate a group signature that contains a single value for each dimension and a group size (k). We then select a set of qualified users using an inverted index that maps from (location, sport, rating) to users. Next, we generate k queries that are compatible with the group signature from the set of qualified users by following distributions described in the workload generation description.

We generate a query's group size as follows. A pre-match group size m is first generated using a Zipfian distribution on the domain $[2, 12]$ with group sizes of size 2 being most frequent. Next, for each query, we generate a random value between 2 and m as the low end of the group size range in that query. We then generate the query group size as $Min(2 * (m - l) + 1, 12 - l + 1)$, where 12 is the maximum group size. This serves to make m the center of each group size range in each prematched query.

In the rare case that we cannot find enough queries to satisfy a group signature, we generate a new group signature and continue. The process terminates when a specified total number of queries have been generated. In order to spread queries from pre-matched groups across the workload, we define a group interval as a parameter that controls randomization. For example, if the group interval is 5000, our workload generator will generate 5000 queries with pre-matched groups at a time and then randomly shuffle all queries within each range of 5000.

To serve as a yardstick, we designed an "optimal" algorithm (OPT) that is only required to scan the workload once and find matches based on pre-matched group id and group size information embedded in the query trace. This information, of course, is ignored by the regular matching algorithms. The match percentage for the "optimal" solution is 100%. The average query latency for OPT varies when the group interval changes. However, it should always be less than the group interval since a match is guaranteed to be found within a group interval in OPT.

6.4 Experimental Results

In this section, we describe and interpret the experimental results to identify the best performing heuristic algorithms. Recall that BMA is the basic matching algorithm that processes the CQS in random order and stops after finding the first match; LMF and HMF are also early stopping algorithms, but they process the CQS in the order of least matchable (respectively highest matchable); finally NES is an algorithm that processes the entire CQS even after finding an initial match to search for larger matches.

We compare these algorithms in terms of system measures (percentage of queries matched and query processing time), and user metrics (average query latency and fairness measured by average group size). Suppose, for example, that a matching algorithm has a group size 2 query it can immediately satisfy, but also a compatible group size 4 query that is compatible but not immediately satisfiable. The algorithm has to exercise forbearance in order to satisfy the size 4 request in the future; greedy choices, by contrast, can have good match percentage but poor fairness.

Performance with Varying Group Interval: Recall that the group interval is a parameter of the workload generator that controls how scrambled the workload compared to a pre-matched set of queries that "seeds" the workload. Clearly, an omniscient, optimal algorithm will be able to match all queries in such a workload because this workload is a randomization of a workload where there is a match for all queries. However, our algorithms such as BMA will do worse than optimal because they are online and not offline, and make heuristic choices to reduce computation. Figure 4 shows the percentage of queries matched by our various heuristics. The hypothetical offline optimal algorithm is not shown because it always achieves 100% matching.

The figure shows that the percentage of matches does not change perceptibly as the group interval (scrambling distance) increases for all algorithms. It also shows NES has the highest match percentage (around 81%) while HMF performs worst (around 76%) and the baseline BMA is slightly better (77%). This is not surprising because NES tries to find the best match without stopping. Intuitively, HMF is bad because it processes highly matchable queries early which makes it less likely that later queries in the CQS will be matched. Given early stopping, LMF does better because it does not squander more matchable queries when other queries will do instead; a random order performs in between LMF and HMF. Note that the difference between 76% and 81% may seem small but if there is revenue attached to each match, a 5% uptick in revenue is appreciable.

NES gets a higher match percentage by processing the CQS more thoroughly. Thus we might hypothesize that NES will require more processing time in return for a higher match percentage. Figure 5 is perhaps surprising because it shows that NES has the smallest average processing time (for example, around 40 μsec at a group interval of 2000) while HMF has close to 50 μsec and the others are in between). Intuitively, this is because NES process more queries on average per scan of the CQS and hence removes queries early from the CQS; this in turn requires less scanning overhead in the future. Note also that the processing time increases slightly with scrambling distance; this makes sense because the more far apart potentially matchable queries are placed in the workload, the greater the average length of the CQS and hence the processing time.

While NES does well from the system provider's point of view, what about from the user point of view? Figure 6 shows the average latency (measured in terms of queries after which a user request is satisfied) for the algorithms. Recall that the workload is generated from a pre-matched set of queries that are scrambled randomly within a group interval. The latency of a query Q for the optimal algorithm is the difference between the index of the last query in the pre-matched set for Q and the index of Q itself. Thus the latency for the optimal algorithm shown as a reference is very

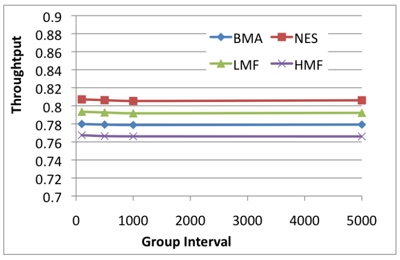


Figure 4: Percentage of matched queries (throughput) as group interval increases.

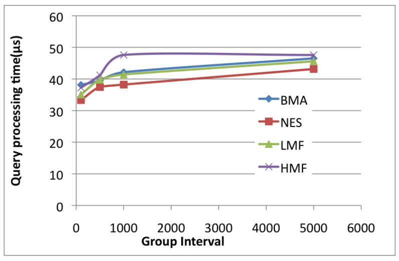


Figure 5: Average query processing time as group interval increases.

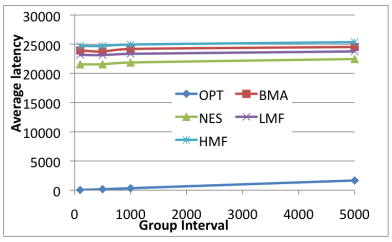


Figure 6: Average query latency as group interval increases.

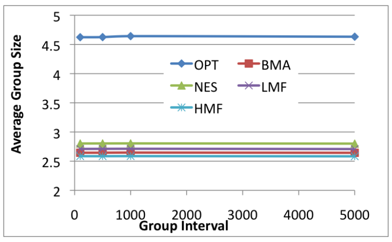


Figure 7: Average matched group size as group interval increases.

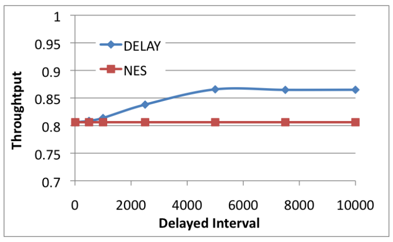


Figure 8: Percentage of matched queries (throughput) as delay interval increases.

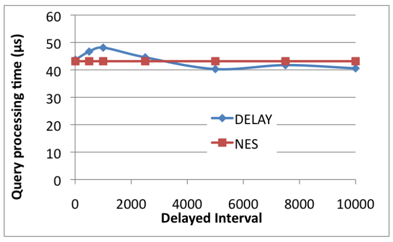


Figure 9: Average query processing time as delay interval increases.

small and increases linearly with the group interval. Once again, NES has the lowest average latency (around 21,000 queries) and HMF has the worst latency (around 25,000).

While NES is the best latency, it is much worse than optimal. Further scrutiny of the results revealed that 60% of queries in NES are fast because NES does find most pre-matched groups. However, because NES is a heuristic, in around 40% of the cases, NES misses finding its pre-generated (and hence optimal) groups. In that case, either the query will never be matched or, by random chance based on the workload, it can be matched later but after a much higher latency whose average value is roughly the average time for a query to find another set of compatible queries. Thus the average latency is pushed very high by outliers. We calculated the median latency for NES at a group interval of 5000 as 135 and the 90% latency as around 7591 compared to the optimal latency of 1658. Given that the outlier latency is largely an artifact of the workload generation model, this artificial increased latency is unlikely to be an issue in practice.

Finally, recall that in measuring fairness, we are trying to penalize algorithms that boost match probability by artificially favoring some group sizes (say small groups) over others. Part of the novelty of our social coordination system is that it allows coordination among users with various size group requests (unlike say a dating service where the sizes are always 2). It would be unfortunate if this generality in the service specification was accompanied by service bias in terms of requested group size. We use the average size of the group when compared to the optimal algorithm as a fairness metric. Figure 7 shows the average group size returned by the various algorithms.

Given the workload, the average group size of matches returned by the optimal algorithm is around 4.6. On the other hand, BMA is around 2.6 and NES is around 2.8. This again follows because NES is prepared to wait to get larger groups and hence is fairer to larger groups.

The results so far show that NES is better than using matchibility as a simple stopping criterion and certainly better than BMA with its random order and early stopping. However, despite this we believe that LMF is a promising heuristic and comes close to NES. For example, if the CQS gets too large, NES may be infeasible and the LMF heuristic may be needed at least for graceful degradation.

Further, LMF particularly shines when there are workloads with a significant fraction of very discriminating users that are hard to satisfy (e.g., want to play tennis in Campbell, CA at 6 am) and very easygoing users (willing to play tennis anywhere in the Bay Area at any time). A small modification of Example 4 can be constructed by adding a fifth query q_5 that can satisfy q_1 and q_2 such that all 5 queries are satisfied by LMF but only 3 are satisfied by NES. Repeating this sequence indefinitely leads to a workload where NES achives a match percentage of only 60% compared to 100% for LMF.

Performance of Delayed Evaluation: So far we have looked at the performance of the basic algorithm augmented with matchibility and no early stopping. We now evaluate the effect of our second and more complex refinement: adding a fixed delay threshold. Recall that in delayed evaluation, for each query, we calculate the average age of other queries in its Compatible Set of Queries (CQS) and delay the processing of that queries if the average age is too small based on simple timer processing. How does the basic performance of all measures change as the delay increases? Intuitively, if the group interval (scrambling interval) is say 5000, delaying processing by around 5000 should make it more likely that a query will behave close to its optimal and the gains should fall off after that.

This is indeed what we find. In all cases, we fix the group interval at 5000 and use the NES algorithm augmented with a delay parameter that varies from 10 to 10,000. Figure 8 shows that as the delay increases the percentage match increases from around 80% for NES (reference line) to around

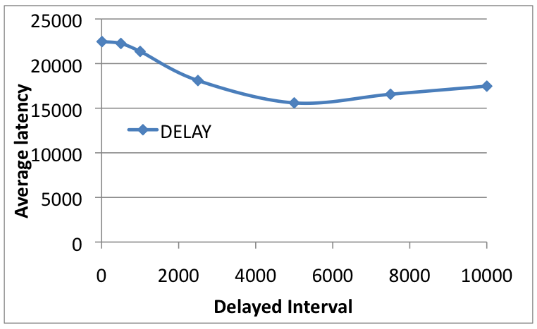


Figure 10: Average query latency as delay interval increases.

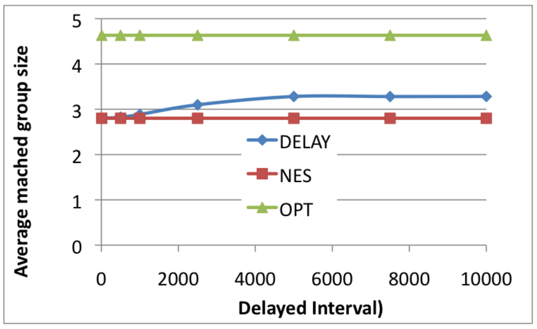


Figure 11: Average matched group size as delay interval increases.

87% but the gains fall off after a delay of 5000. Figure 5 shows that the query processing time can increase with delay from around 40 μsec to around 50 μsec because of the overhead of the delay processing. However, as we get closer to the optimal delay of 5000, the processing time falls back to around 40 μsec and is even slightly faster than NES. We hypothesize that the better match efficiency makes up for the increase in delay timer processing (which is a fixed overhead regardless of the magnitude of the delay).

Figure 10 shows a more interesting tradeoff. As the delay increases, we see that the latency drops sharply at first from 23,000 to around 15,000. This is because increasing the match percentage by 5% reduces the outliers by 5% which sharply decreases the average latency, Beyond the optimal delay of 5000, however, the match percentage does not increase and the delay is merely an artificial waiting penalty: thus beyond 5000 the average latency starts increasing again.

The user will rightly complain that this is cheating. In a synthetic workload, we knew the group interval and hence could estimate the optimal delay. However, in a real deployment, the system can keep statistics such as Figure 10 by periodically varying the delay (akin to explore-exploit systems) to find the knee of the curve. The knee of the curve can be used to estimate the optimal amount of delay to be added.

Finally, Figure 11 shows that delayed processing also decreases the bias against large groups by coming closer to the average group size of the optimal algorithm. As the delay increases to the knee of the curve (5000), the average group size gets close to 3.5 which is much closer to 4.5 which represents the ideal (optimal) compared to say 2.6 for BMA.

Recommendations for Deployment: Our results suggest that a combination of DELAY and NES performs well across all measures. The system can find the smallest value of added delay after which matching performance does not improve significantly. LMF may be useful in workloads that have many discriminating users, and when the average CQS length become so long that NES becomes infeasible.

In terms of scaling, an average of 40 μsec implies 25,000 queries a second on a single core using a high end 2.5 Ghz machine. However, throughput can be increased by forking multiple threads, leveraging multiple cores, and parallelizing query processing by region using multiple servers: users in Cairo are unlikely to coordinate with users in Cupertino.

7. DISCUSSION

Thus far, we have proposed a new query concept called enmeshed queries for fluid social coordination and shown that it has a scalable implementation. We now step back and consider the role enmeshed queries can play in the wider world of social coordination referred to in the introduction. When we consider systems like Evite and Meetup, we are immediately led to distinguish between *user selected* and *system selected* coordination. In enmeshed queries, the user declares his preferences and the system selects a compatible set of other users. By contrast, in Evite the user selects the group; and the system assists with further details of coordination such as RSVPs. Meetup is more interesting; the users declare intent by specifying place (zipcode) and activities (keywords), and the system suggests a list of groups that match the user intent. While the system *assists* the user by making suggestions, the user ultimately selects and joins the desired group. Meetup suggests another related distinction between *browsing* and *solving*. Enmeshed queries are akin to Information Retrieval (IR) methods; browsing as in Meetup is akin to search engines like Google and Bing. Considered from this lens, even Foursquare and Facebook Places are browsing approaches followed by user selection. Foursquare allows a user to query for friends in a given place, and for the places where a given set of friends are, presenting the results in visual form. The user then selects the coordination by physically going to a location to meet friends.

While browsing models have several advantages, they have some issues as well in our general model. First, browsing models have to deal with the threat of thrashing: it may be that user A likes a group with user B, but user B does not like User A, making it hard to drive coordination to closure. Second, browsing models expose a great deal of information about user behaviors to the wide world, raising privacy concerns. Third, browsing models can lead to redundancy and inefficiency. In Meetup, for example, there could be duplication of interests across meetings: if the system were to detect such duplication, one would have algorithmic challenges similar to that of enmeshed queries.

Despite the differences between user selection and system selection, there are some concepts that seem fundamental to coordination in general. First, the concept of fundamental coordination selection attributes such as location, times, people and activities seems to be universal. Imagine an application that generalized Evite, Foursquare, and Meetup by having various areas of the screen for these fundamental attributes and where each application is simulated by specifying values for some attributes and not for others. Second, the implementation concepts PINDEX (index over a subset of selective attributes), PQS (the list of other users/queries that partially match a user's attributes), and CQS (the short list that matches a user's coordination desires) seem to be useful for both browsing and system selected models. Third, group constraints may be generally useful; Meetup has a simple cardinality constraint today in terms of a maximum group size, but users may find other group constraints to be helpful to limit scope.

It should be clear from this discussion that existing systems like Meetup, Evite, and Foursquare follow a different model (user selected, browsing) and hence cannot be modeled by enmeshed queries (system selected, solving). We believe that both forms of coordination are useful and should be combined in an ideal coordination platform. We have designed an initial version of such a platform with APIs for both user selection (Create, Respond, Finalize) as well as for system selection via enmeshed queries (DeclareInterests). Applications could well combine these APIs: for example, an application may use system selection to *suggest* an initial set of users but to formalize the coordination using *user selected* measures akin to Evite.

8. CONCLUSIONS

Social *coordination* may represent the next step beyond the social *awareness* provided by today's Online Social Networks (OSNs). We focused on the problem of fluid social coordination where the set of people involved is unknown at the start, may change quickly over time, and may *not* be part of one's friends in any OSN. Such fluid coordination allows forming weak ties [4] that can enrich our lives beyond the strong ties formalized by OSNs.

In our formalism, users declaratively specify coordination objectives using selection constraints on coordination variables, join constraints on pairs of user variables, and group size constraints. While declarative specification can reduce user effort compared to browsing, we recognize that, in some cases, browsing can allow more flexibility. We suggest the following research directions in social coordination:

1. Economics: Enmeshed queries extend dating services to arbitrary group sizes. Some dating services provide better matches for users who pay more. What is the natural way to extend enmeshed queries to specify a willingness to pay for matches? How does this relate to auction theory?

2. Soft Constraints: While we consider hard coordination constraints, users might prefer to declare relative preferences over attributes like place and time. How can enmeshed queries be extended to handle such "soft-constraints"?

3. Recommendations: Amazon and NetFlix recommend new choices based on past selections. On what basis should a coordination system recommend groups to users? A user that often plays tennis may like to hear about 3 nearby, compatible tennis players who wish to find a fourth player.

4. Query Flexibility: Our simple model of a conjunction of disjunctions can be generalized. For example, in ad matching [2] a richer set of boolean queries can be expressed at the possible cost of increased computation time. Further, our paper uses a completely declarative (system chooses) model and allows no browsing (user chooses) as in Meetup. What is the best way to combine browsing and declaration?

5. Data Mining: Social coordination tools may allow social scientists to answer questions about the sociology of coordination. What are the metrics (e.g., fraction of successful coordinations, keystrokes to coordination completion) by which one judges a successful social coordination? A social scientist could test hypotheses about successful coordinations such as "small groups are more likely to find matches".

Social coordination fulfills a basic human need to connect to other human beings. Further, fluid social coordination allows building weak ties to a larger set of people than our friends. While such weak ties classically occur by serendipity as in meetings at the proverbial water cooler, enmeshed queries may help institutionalize such serendipity. While we have taken a small step in formalizing aspects of the problem, the vista for further work appears inviting: the solution space can be enriched if social scientists, database theorists, economists, and system designers all join the conversation.

9. ACKNOWLEDGMENTS

We would like to thank Adam Silberstein, whose earlier work on user selected coordination inspired our work on enmeshed queries. We also thank Raghu Ramakrishnan and Rakesh Agrawal for providing us useful feedback.

10. REFERENCES

[1] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for internet databases. In *SIGMOD*, 2000.

[2] M. Fontoura, S. Sadanandan, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, and S. V. J. Y. Zien. Efficiently evaluating complex boolean expressions. In *SIGMOD*, pages 3–14, 2010.

[3] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.

[4] M. Granovetter. The Strength of Weak Ties. *The American Journal of Sociology*, 78(6):1360–1380, 1973.

[5] N. Gupta, L. Kot, S. Roy, G. Bender, J. Gehrke, and C. Koch. Entangled queries: enabling declarative data-driven coordination. In *SIGMOD*, 2011.

[6] L. Kot, N. Gupta, S. Roy, J. Gehrke, and C. Koch. Beyond isolation: research opportunities in declarative data-driven coordination. *SIGMOD Record*, 39(1):27–32, 2010.

[7] T. Lappas, K. Liu, and E. Terzi. Finding a team of experts in social networks. In *KDD*, pages 467–476, 2009.

[8] N. Lynch and M. Merritt. Introduction to the Theory of Nested Transactions. *Theoretical Computer Science*, 1(62):123–185, 1988.

[9] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. In *ICDE*, 2006.

[10] A. Machanavajjhala, E. Vee, M. N. Garofalakis, and J. Shanmugasundaram. Scalable ranked publish/subscribe. In *VLDB*, 2008.

[11] K. Mamouras, S. Oren, L. Seeman, L. Kot, and J. Gehrke. The complexity of social coordination. In *VLDB*, 2012.

[12] J. a. Pereira, F. Fabret, F. Llirbat, R. Preotiuc-Pietro, K. A. Ross, and D. Shasha. Publish/subscribe on the web at extreme speed. In *VLDB*, 2000.

[13] L. Sweeney. k-Anonymity: A Model for Protecting Privacy. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 5(10):557–570, 2002.

[14] D. B. Terry, D. Goldberg, D. A. Nichols, and B. M. Oki. Continuous queries over append-only databases. In *SIGMOD*, 1992.