Optimizing Human Computation to Save Time and Money

MSR-TR-2014-145

Benjamin Livshits
Microsoft Research

George Kastrinis
University of Athens

Abstract

Crowd-sourcing is increasingly being used for providing answers to online polls and surveys. However, existing systems, while taking care of the mechanics of attracting crowd workers, poll building, and payment, generally provide little by way of cost-management (e.g. working with a tight budget), time-management (e.g. obtaining results as quickly as possible), and controlling the margin of error (e.g. working on a sample population which is largely different from the general census statistics). The problems above create significant pain points for those wanting to run large-scale surveys, such as people doing polling for political campaigns, marketing professionals, and the like.

Our work unlocks the possibility of large-scale polling on a budget though the use of novel *optimization strategies*. Our work, is based on InterPoll, a platform for programming crowdsourced polls. In this paper, we present three static and three runtime optimizations for InterPoll polls represented as LINQ queries. The former share some similarities for traditional compiler optimizations, while the latter borrow insight from databases and real-life polling strategies.

These optimizations lead to significant improvements in practice. In our experiments we observed tenfold savings in survey cost and time savings of as much as 20 hours for some of the queries.

1. Introduction

Online surveys have emerged as a powerful force for assessing properties of the general population, ranging from conducting marketing studies, to product development, to political polls, to customer satisfaction surveys, to medical questionnaires. Online polls are widely recognized as an affordable alternative to in-person surveys, telephone polls, or face-to-face interviews. Psychologists have argued that online surveys are far superior to the traditional approach of finding subjects in the college student pool, leading to the famous quip about psychology being the study of the college sophomore [6].

Online surveys allow one to reach wider audience groups and to get people to answer questions that they may not be comfortable responding in a face-to-face setting. While online survey tools such as Instant.ly, SurveyMonkey, Qualtrics, and Google Customer Surveys take care of the mechanics of online polling and make it easy to get *started*, the results they produce often create more questions than they provide answers [7–9, 12, 15, 33]. Indeed, polling too few yields results that are not statistically significant; polling too many is a waste of money. Prior work [22] has focused on deciding how many people to poll to get statistically representative responses. Surveys, both online and offline, suffer from *selection biases*, to mitigate which InterPoll supports automatic unbiasing.

Our work in this paper is based upon InterPoll, a crowd-sourced survey building tool [21]. One of the goals of InterPoll is to make running crowd-sourced polls easy for the developer. InterPoll accomplishes this by using LINQ [25], language-integrated queries. [1] These queries are translated into surveys that are run on a general-purpose crowd backend, such as Amazon's Mechanical Turk. InterPoll provides a runtime that effectively integrates human and machine computation. This paper showcases some of the optimizations this platform enables.

1.1 Motivating Examples

One of the goal of InterPoll is to make running crowd-sourced polls easy for the developer as well as make them easy to integrate into existing bodies of code.

Example 1 (Liberal arts majors) A simple poll may be performed the following way:

```
1  var people = new MTurkQueryable<Person>(true, 5, 100, 2);
2  var liberalArtsPairs = from person in people
3        where person.Employment == Employment.STUDENT
4        select new {
5              Person = person,
6              Value = person.PoseQuestion<bool>(
7                    "Are you a liberal arts major?")
8              },
9        Income = person.Income;
```

The first line gets a handle to a population of users, in this case obtained from Mechanical Turk, although other back-ends are also possible. Populations on which we operate have associated demographic information; for example, note that the `where` clause on line 3 ensures that we only query (college) students. This poll will ask (college) students if they study liberal arts, producing an iterator of ⟨`Student`, `bool`⟩ pairs represented in .NET as `IEnumerable`. □

Example 2 (Counting) Given `liberalArtsPairs`, it is possible to do a subsequent operation on the result, such as printing out all pairs or using, the `Count` operation to count the liberal arts majors:

```
var libralArtMajorsCount =
      (from pair in liberalArtsPairs
```

[1] LINQ is natively supported by .NET languages, with Java providing similar facilities with JQL.

```
3            where pair.Value ==true
4            select  person).Count();
5    var percentage =100.0*libralArtMajorsCount/ liberalArtsPairs .Count();
```

Lines 5 and 6 compute the percentage of liberal art majors within the previously collected population. □

Example 3 (Uncertainty) InterPoll explicitly supports computing with uncertain data, using a style of programming proposed in Bornholt *et al.* [4].

```
1    var liberalArtWomen =from person in people
2      where person.Gender ==Gender.FEMALE
3      where person.Employment ==Employment.STUDENT
4      select  person.PoseQuestion<bool>("Are you a liberal  arts  major?");
5
6    var liberalArtMen  =from person in people
7      where person.Gender ==Gender.MALE
8      where person.Employment ==Employment.STUDENT
9      select  person.PoseQuestion<bool>("Are you a liberal  arts  major?");
10
11   var femaleVar =femaleSample.ToRandomVariable();
12   var maleVar =maleSampleList.ToRandomVariable();
13    if (femaleVar > maleVar){
14     Console.WriteLine("More female  liberal   arts  majors.");
15   }else{
16     Console.WriteLine("More male  liberal   arts  majors.");
17   }
```

Here, we convert the Boolean output of the posted question to a random variable (lines 11 and 12). Then we proceed to compare these on line 13. Note that the implicit `>` comparison on line 13 actuality compiles to a *t-test* on `femaleVar` and `maleVar`. □

1.2 High-Level Optimization Goals

In optimizing queries in InterPoll, we try to satisfy the following goals, in order of general importance.

- Reduce **overall cost** for running a query; clearly for many people, reducing the cost of running survey is the most important "selling feature" when it comes to optimizations. Not only does it allow people with a low budget to start running crowd-sourced queries, it also allows survey makers to 1) request more samples and 2) run their surveys more frequently. Consider someone who may previously have been able to run surveys *weekly* not able to do so *daily*.
- Reduce the **end-to-end time** for running a query; we have observed that in many cases, making surveys requires *iterating* on how the survey is formulated. Clearly, reducing the running times allows the survey maker to iterate over their surveys to refine the questions much faster. Consider someone who needs to wait for week only to discover that they need to reformulate their questions and run them again.
- Increase the precision and reduce the **error rate** (or confidence interval) for the query results; while we support unbiasing the results in InterPoll, one of the drawbacks that is often cited as a downside of unbiasing is that the *error rate* goes up. This is only natural: if we have an unrepresentative sample which we are using for extrapolating the behavior for the overall population, the high error rate will capture the paucity of data we are basing our inference on.

This paper focuses on a mix of both static and runtime, feedback-driven optimizations targeting the three goals above. We note that unlike many compiler or runtime optimizations, savings in this paper are of the 10× variety, not the 10% variety[2].

1.3 Contributions

Our contributions consist of optimizations, designed to fulfill the three goals outlined above.

- We propose three static optimizations of LINQ trees;
- We implement optimizations for separating *qualifying questions* and running queries in stages so as to reduce the cost of surveys with a low yield;
- We develop strategies for re-balancing queries that require considering *pairs* of people to reduce the amount of wasted effort and decrease the completion time;
- We automate the process of creating *representative panels* in order to reduce the margin or error when unbiasing surveys and to minimize the amount of wasted effort.
- We evaluate these ideas experimentally on surveys that involve hundreds of crowd workers. Our optimizations lead to significant improvements in practice. In our experiments we observed tenfold savings in survey cost and time savings of as much as 20 hours for some of the queries.

1.4 Paper Organization

The rest of this paper is organized as follows. Section 2 presents *static optimizations* that involve requiring LINQ query trees. The next three sections present and evaluate individual runtime optiomizations. Since evaluations tend to be fairly different, we found this approach easier to understand than having an overall evaluation section at the end of the paper. Section 3 focuses on *dynamic execution strategies* that optimize for the query *yield*. Section 4 presents optimizations designed to re-balance decision queries that require considering *pairs* of people. Section 5 presents optimizations designed to pre-compute a balanced panel. Section 6 gives an overview of related work and Section 7 concludes. To help with exposition, throughout this paper, we decided to co-locate descriptions of optimizations with experimental results.

2. Tree Rewriting Optimizations

One of the reasons we chose Language-Integrated Queries (LINQ) in .NET was the support for intricate *providers*, which allow the semantics of query processing to be redefined in significant ways. Additionally, LINQ queries seamlessly connect SQL-like queries over human-generated data with the rest of the programs with all its traditional constructs such as `ifs`, `whiles`, function calls, etc. Figure 4 shows some examples where regular constructs such as boolean tests and arithmetic operations are embedded within (and potenially outside) LINQ code.

[2] Lastly, we also aim to reduce *potential frustration* on the part of the workers. While it is possible to create aggressive evaluation strategies using InterPoll, doing so may result in a certain amount of push-back from the workers and a loss of reputation for InterPoll. One way to measure worker sentiment is via sites such as various discussion fora such as HITsWorthTurkingFor (`http://www.reddit.com/r/HITsWorthTurkingFor/`) or sites like `www.turkernation.com`, which aim to highlight easy and well-paying HITs (human interest tasks) and warn others about hits that, for instance, do not pay as well as advertised or require more time than expected to complete. Throughout the experiments described in this paper, we aimed to remain "good citizens."

```
collection   ::=  from var in collection
                  [where boolExpr] select getter
collection   ::=  from var in collection
                  [where boolExpr] select creation
collection   ::=  Set
getter       ::=  var.field | var.getter(args)
creation     ::=  new struct
struct       ::=  {alias[, ...]}
alias        ::=  name = getter
boolExpr     ::=  true | false | getter compOp const |
                  boolExpr logicalOp boolExpr | !boolExpr
compOp       ::=  == | != | > | >= | < | <=
const        ::=  "..." | 0, 1, ...|...
logicalOp    ::=  && | ||
```

Figure 1: Small BNF for LINQ expressions which InterPoll can optimize.

```
from structure in
    from person in employees
    where person.Wage > 4000 && person.Region =="WA"
    select new {
        Name = person.Name,
        Boss = person.GetBoss(),
        Sales = person.GetSales(42)
    }
where structure.Sales.Count > 5
select structure.Boss;
```

```
from person in employees
where (person.Wage > 4000 && person.Region =="WA")
      && (person.GetSales(42).Count > 5)
select person.GetBoss();
```

(a) Single-level flattening.

```
from structure in
    from person in employees
    where person.Wage > 4000 && person.Region =="WA"
    select new {
        Name = person.Name,
        Boss = person.GetBoss(),
        Sales = person.GetSales(42),
        Winter = person.Q1 + structure.Q2,
        Summer = person.Q3 + structure.Q4
    }
    where structure.Sales.Count > 5
    select Employee.GetMin(structure.Winter, structure.Summer);
```

```
from person in employees
where (person.Wage > 4000 && person.Region =="WA")
      && (person.GetSales(42).Count > 5 && person.Q1 > 10)
select person.Q4;
```

(b) Multi-level flattening.

Figure 2: Tree flattening optimizations illustrated.

LINQ expressions & LINQ providers: LINQ provides an easy way to programmers for accessing the internal parts in each LINQ query. That is possible via LINQ expressions. Each LINQ query is translated into a expression ASTs, which can be rewritten by LINQ Providers. This is how InterPoll operates: by providing an appropriate set of visitors to rewrite LINQ query trees to both optimize them and also connect query trees to the actual data which comes from Mechanical Turk. The latter kind of "plumbing" is responsible for obtaining Mechanical Turk data in XML format and then at runtime parsing and validaing it, and embedding the data it into type-safe runtime data structures. The process of expression tree rewriting is done via visitors whose job is to override one of more methods in the visitor parent class.

```
protected override Expression VisitMethodCall
    (MethodCallExpression node)
{
    var methodName =node.Method.Name;
    switch(methodName) {
        case "Select" :
            ...
            return <newly−constructed−node>;
        case "Where" :
            ...
            return <newly−constructed−node>;
    }
}
```

Figure 3: Rewriting LINQ expression trees.

A schematic example of such a method is shown in Figure 3.

We should also keep in mind that LINQ query evaluation is inherently *lazy*. Using different LINQ providers allows us to decide on an evaluation strategy at runtime or change evaluation strategies as we get more data based on profiling.

Figure 1 shows the subset of LINQ to which our optimizations in this section apply. While other aspects of LINQ will run just fine, we do not attempt to optimize them. In the rest of this section, we discuss individual tree rewriting-based optimizations.

2.1 Query Flattening

To build one's intuition, some examples of flattening in action are shown in Figure 4. There are three examples of flattening, going from the simplest to the most complex. The complexity is determined by both the level of nesting and the operations involved. This example concerns itself with people and and their quarterly earnings (`Q1`–`Q4`). Flattening involves getting rid of inner structures by "expanding them out." For instance, using the aliases `{Sales = person.GetSales(42), Boss = person.GetBoss()}` from the original query in Figure 2a becomes

```
from person in employees
where person.Wage > 4000 && person.Region ="WA"
where person.GetSales(42).Count > 5
select person.GetBoss();
```

which yields the outcome shown at the bottom of Figure 2a after combining the two `where` clauses.

The algorithm in Figure 4 describes the process. The trivial base case is when the LINQ expression is of depth one. Otherwise, the inner expression in the `from` clause is first flattened recursively, and then the process repeats for the outer layer. The goal of the flattening process is to remove intermediate structures that are created. Those anonymous structures can only define fields, and, as such, the replacement process is focused on expressions with fields as in `var.field`. Each such expression is replaced by effectively inlining expression used in the definition of that field.

The end result is a new LINQ expression where the `from` clause is that of the inner expression, the `where` clause is the logical AND combination of the `where` clause from the inner expression and the outer one (after all the replacements), and the `select` clause is the outer one (after all the replacements).

```
function Flatten($LinqExpr\ L$)
Returns: $LinqExpr\ L'$

1: // Base case
2: if $L.Depth = 1$ then
3:    $L' = L$
4: else
5:    $Inner = Flatten(L.From)$
6:    $OuterWhere = L.Where$
7:    $OuterSelect = L.Select$
8:    // New types can define fields but not functions
9:    for all $var.field \in OuterWhere \cup OuterSelect$ do
10:      if $Inner.Defines(\texttt{typeof}\ var)$ then
11:         // Get the expression that initializes field
12:         $Replacement = Inner.Select.ExprFor(field)$
13:         $Swap(var.field, Replacement)$
14:      end if
15:   end for
16:   // OuterWhere and OuterSelect have now been altered
17:   $L'.From = Inner.From$
18:   $L'.Where = Inner.Where \wedge OuterWhere$
19:   $L'.Select = OuterSelect$
20: end if
```

Figure 4: Algorithm Flatten for flattening a LINQ expression.

```
function SplitQuery($LinqExpr\ L$)
Returns: LinqExpr $L'$

1: $AllQuestions = L.Questions$
2: $AllDemographics = L.Demographics$
3: $Flat = Flatten(L)$
4: $FilterQuestions = Flat.Where.Questions$
5: $FilterDemographics = Flat.Where.Demographics$
6: $NewStruct = new\{$
7:    $Questions = AllQuestions \setminus FilterQuestions,$
8:    $Demographics = AllDemographics \setminus FilterDemographics\}$
9: $L'.From = Flat.From$
10: $L'.Where = Flat.Where$
11: $L'.Select = NewStruct$
```

Figure 5: Algorithm SplitQuery for splitting questions and demographics in a LINQ expression.

2.2 Query Splitting

The goal of query splitting is to rewrite a LINQ expression in such a way where general and demographic questions that filter based on demographic charasteristics are gathered in the `where` clause, and all the rest are present in the `select` clause. For instance, the end-result might look like the following:

```
from person in people
where person.PoseQuestion(" ... ") > 100 && person.Gender =="MALE"
 select new {
          Q1 =person.PoseQuestion(" ... "),
          Q2 =person.PoseQuestion(" ... "),
          Education =person.Education
}
```

The algorithm in Figure 5 describes the process. First, the set with all the general and demographic comparison present in the LINQ expression is created. Afterwards, the expression is flattened so that there is only one `where` clause and from that a second set of (filter) questions is created. In the final LINQ expression, the `select` clause only has the questions present in the initial expression that are not already handled in the `where` clause.

2.3 Common Subexpressions Elimination

Finally, Common Subexpressions Elimination (CSE) aims, given two LINQ expressions, to identify common subexpressions and then merge the inputs in such a way that each LINQ expression references the same common subexpression (instead of having two different occurrences). The optimization is modeled on its counterpart in standard compiler literature [1]. However, the matching is done on the structure of LINQ subtrees. Equality is established by computing hash functions. Computing the hash values for leaf nodes is trivial, since they will hold constant values. Computing the hash values for internal nodes is done by combining the hash values of every children node.

```
function CommonSubexpressions($LinqExpr L_1$, $LinqExpr L_2$)
Returns: $LinqExpr\ L'_2$

1: // Rewrite both expressions normalizing variable names
2: $L_1 = NormalizeVariableNames(L_1)$
3: $L_2 = NormalizeVariableNames(L_2)$
4: // Create Hash Maps, mapping each subexpression to an ID
5: $H_1 = CreateHashes(L_1)$
6: $H_2 = CreateHashes(L_2)$
7: // Gather expressions with the same ID
8: $Common = H1 \cap H2$
9: // Expr1 in L1 has the same ID with Expr2 in L2
10: for all $e \in Common$ do
11:    $e_1 = H_1[e.Key]$
12:    $e_2 = H_2[e.Key]$
13:    $Replacements[e_1] = e_2$
14: end for
15: // Using the mapping above, replace subexpressions in L2 with
    ones from L1
16: $L'_2 = MakeReplacements(Replacements, L_2)$
```

Figure 6: Algorithm CommonSubexpressions for merging common parts in two LINQ expressions.

The algorithm in Figure 6 describes the process. First, there is a preprocessing step where variable names are normalized in each LINQ expression. Afterwards, each expression is traversed and each subexpression is assigned an ID is such a way that syntactically equivalent expressions will have the same ID assigned to them. Subsequently, those IDs are used in order to identify common subexpressions that can be afterwards merged so there is only on occurrence in both initial LINQ expressions.

2.4 Summary

This section has described three different static optimizations that involve LINQ tree rewriting. In addition to providing actual optimizations, these serve as a way to both prepare the reader for more complex optimizations in the rest of the paper and also to normalize queries for further optimizations. Subsequent sections will address various runtime optimizations.

3. Yield Optimizations

One of the key challenges in many surveys is the problem of *low yield*. While we can poll a large number of people, only some meet our filter. For instance, we may be interested in only females or only in people who are employed full-time and over the age of 40. Or we may be interested in surveying those who are iPhone users. All of these are examples of *qualifying questions*. These are widely used in marketing, for example, where one is interested in only a well-defined group and its response to a particular product, for instance. When it comes to query performance, having narrow qualifications both creates queries that both take a great deal of time and are costly. In this section, speaking broadly, we aim to make these low-yield queries efficient.

```
var query =from person in people
select new {
    Hired =person.PoseQuestion<bool>(
            "Have you started a new job in the last 6 months?"),
    HiredFriends =person.PoseQuestion<bool>(
            "Have any of your friends or
             neighbours started a new job in the
             last 6 months?"),
    Fired =person.PoseQuestion<bool>(
            "Have you quit or lost a job in the
             last 6 months?"),
    FiredFriends =person.PoseQuestion<bool>(
            "Have any of your friends quit or
             lost a job in the last 6 months?"),
    Gender =person.Gender, Employment =person.Employment,
    Education =person.Education,
};
query =from person in query where
    person.Gender ==Gender.FEMALE &&
    person.Education ==Education.BACHELORS_DEGREE
select person;
```

(a) Jobs query.

```
var query =from person in people
select new {
    Hours =person.PoseQuestion(
        "In the past month, how many hours of sleep
         have you gotten each night on average?",
        "More than 10 hours", "8−10 hours", "6−8 hours",
        "4−6 hours", "4−6 hours", "less than 4 hours"),
    WhatToDo =person.PoseQuestion(
        "Generally, when you feel yourself getting tired
         during the day, what do you primarily do to counteract
         this feeling?", "Drink coffee", "Drink an energy drink",
        "Get a healthy snack/shake", "Take vitamins", "Exercise",
        "Take a nap", "Nothing at all", "I do not get drowsy
         during the day", "Other"),
    Gender =person.Gender,
    Employment =person.Employment,
};
query =from person in query where
    person.Employment ==
Employment.WORKED_FULL_TIME_YEAR_ROUND
select person;
```

(b) Sleeping habits query.

```
var query =from person in people
select new {
    Activity =person.PoseQuestion(
            "What is currently preventing you
             from being more physically active?",
            "Time Available", "Desire and Motivation",
            "Weather", "Laws of Thermodynamics",
            "Physical Disability"),
    Gender =person.Gender, Income =person.Income,
    Employment =person.Employment, Age =person.Age,
};
query =from person in query where
    person.Age >= 30 && person.Employment ==
Employment.WORKED_FULL_TIME_YEAR_ROUND
select person;
```

(c) Physical activity query.

```
var query =from person in people
select new {
    ShopOnline =
        person.PoseQuestion<bool>("Do you shop online?"),
    PercentageOnlineShopping =person.PoseQuestion<int>(
        "If so, how much of your holiday
         shopping do you do online (percentage)?"),
    MoreOrLess =person.PoseQuestion(
        "Do you plan to shop online more or less?",
        "More", "Less", "Same"),
    Gender =person.Gender, Education =person.Education,
    Income =person.Income, Employment =person.Employment,
};
query =from person in query where person.Gender ==Gender.MALE
select person;
```

(d) Online shopping query.

Figure 7: Queries for comparing the three evaluation strategies.

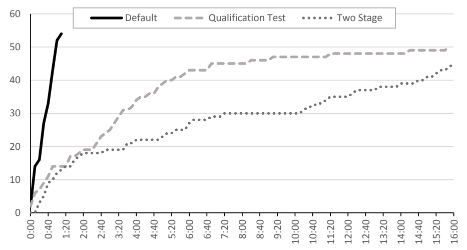
3.1 Evaluation Strategies

The default approach of post-filtering once the data is obtained is clearly wasteful for low-yield queries, as we pay for everyone who takes the poll, not only for those whose completes are useful to us. If the yield is 10%, we should expect to pay ten-fold for this kind of query. In this section we propose three strategies for running surveys with qualifying questions. We assume that the query has been pre-processed using the query splitting optimization in Section 2.2.

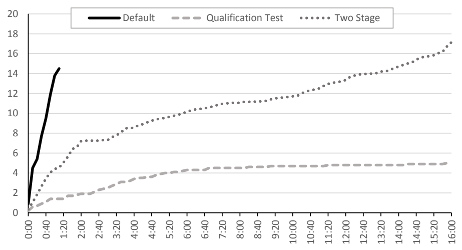
Each execution strategy is responsible for creating a survey, polling for results, making dynamic changes to the survey (e.g. changing the amount of people that take part or the amount they are paid) and finally presenting the results to the user. During execution, runtime metrics are recorded. For instance, the yield of the query or the times when each person took the survey. Each execution strategy is also supplied with an execution policy, which is responsible given the runtime metrics gathered so far to make decisions about the rest of the execution. For instance, how many more people to ask or if the reward amount should change.

1. The **default** strategy involves asking a larger number of workers and then post-filtering to only include those which match our qualification criteria.
2. A **two-stage** survey strategy involves separating our HIT into two: the first tests to see if a worker matches our qualification. If so, she is directed to our second-stage HIT to complete the rest of the survey. Through internal bookkeeping, we ensure that an unqualified worker will not be able to accept the second-stage HIT. We use Amazon's MECHANICAL TURK API to notify (via email) workers that qualified the first-stage. The advantage of this approach is our ability to balance the payoff in the two stages. For instance, we can pay quite little for the first-stage qualifying task and we can pay quite a bit more for those who qualify to take the second-stage task. Note that the questions in the first stage can be quite arbitrary. For instance, we can ask workers to capture their attitude towards a presidential campaign in 200 characters or less and then perform *sentiment analysis* to decide if these users qualify for the next stage of the survey.
3. The last **qualification-based** strategy involves using a built-in qualification mechanism in MECHANICAL TURK, which allows one to qualify users based on multiple-choice questions. While this approach is less general than the two-stage strategy above, the key advantage is that we are *not* required to pay for users who fail to qualify.

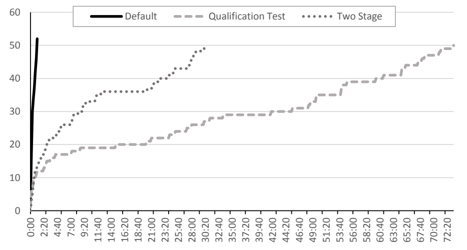
All of three strategies above can be captured by the following cost equation, which summarizes the expected cost of



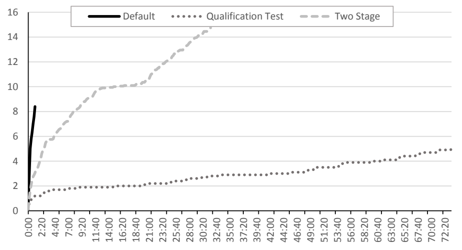
(a) Job query in Figure 7a.



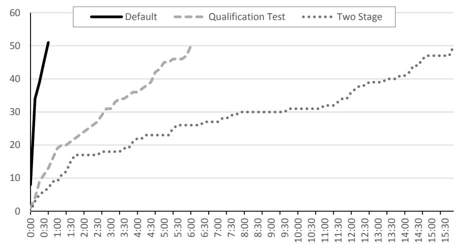
(b) Job query in Figure 7a.



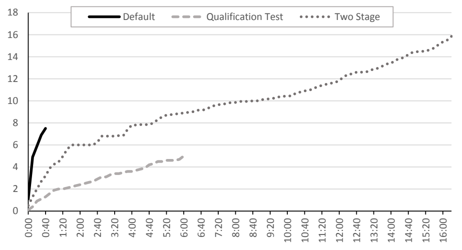
(c) Workout query in Figure 7c.



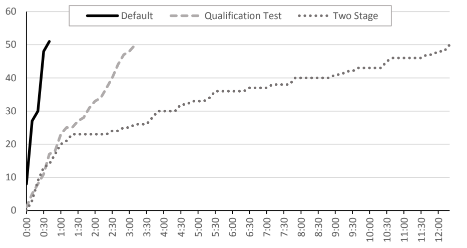
(d) Workout query in Figure 7c.



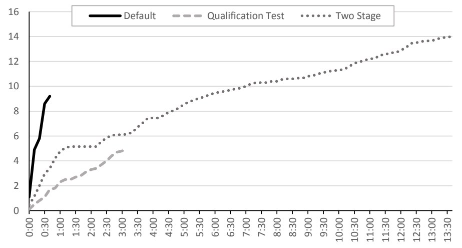
(e) Online shopping query in Figure 7d.



(f) Online shopping query in Figure 7d.



(g) Sleep query in Figure 7b.



(h) Sleep query in Figure 7b.

Figure 8: Completion times and money spent for different evaluation strategies. Time is shown on the x axis.

obtaining N qualified workers:

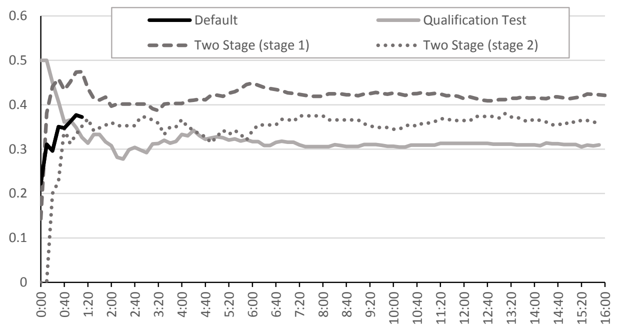
$$E[C_N] = N \times \left[\frac{c_1}{y_1} + \frac{c_2}{y_2}\right]$$

where c_1 is the first-stage cost per person and c_2 is the second-stage cost per person who reaches that stage. Finally, y_1 is the yield, which we for now assume to be independent of c_1, c_2, or other parameters and also constant, i.e. not changing with as we continue running the survey or depending on the hour of the day. y_2 is the yield for the second stage: how many people of the ones eligible for the second stage complete it.

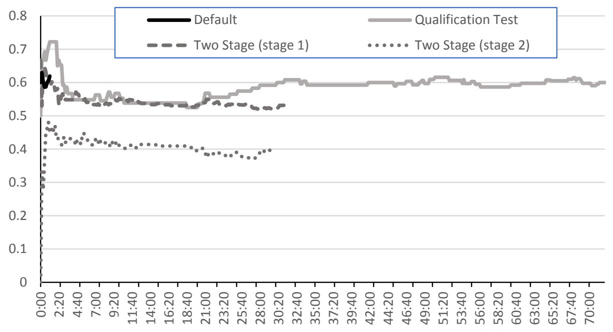
For the default strategy, there is no second stage, so the equation reduces to $E[C_N] = N \times c_1/y_1$. For the qualification-based strategy, there is no extra cost to the first stage, so the equation reduces to just $E[C_N] = N \times c_2/y_2$. For the two-stage strategy, the win may be not as clear and is dependent on the yields. Note that the second-stage yield can vary greatly: we see it ranging between as low as 34% and as high as 90% in our experiments.

3.2 Evaluation

To compare the effectiveness of three execution strategies in Section 3.1, we have selected four representative InterPoll queries, shown in Figures 7a–7b. All of these queries have a relatively broad range of qualifications, i.e. we ask for females with a bachelor degree or higher (Figure 7a) or people who work full-time (Figure 7b). The queries focus on issues of broad appeal ranging from sleep habits to online shopping.



(a) Yield for the jobs query in Figure 7a.



(b) Yield for the workout query in Figure 7c.

Figure 9: Yield for two queries with different evaluation strategies. Time is shown on the x axis.

Results: Figure 8 shows the completion times and the money spent by each of the three strategies, on the left and right, respectively.

- The qualification test strategy is consistently cheaper than the alternatives. Contrast \$8.4 vs. \$1.2 at 1 hour and 10 minutes, a ratio of 7× in Figure 7c. Also contrast \$14.5 vs. \$1.4 also at 1 hour and 10 minutes, a ratio of 10× in Figure 7a. Somewhat surprisingly, the two-phase strategy is also more expensive than the default in all of our tests.
- The default, one-stage strategy completes considerably faster than the more frugal two-stage or qualification test-based approaches. Contrast 40 minutes vs. 16 hours in Figure 8b or 50 minutes vs. 12 hours in Figure 8h.
- In summary, the qualification test is preferable if the end-to-end cost is of the essence. Otherwise, the default strategy generally delivers results faster.

Figure 9 shows the details of the yield for two of representative queries of the four we showed in Figure 8. Overall, the yields are comparable, while the compeletion time and costs differ, as indicated above.

```
var query =from person in people
select new {
      Attitude =person.PoseQuestion(
         "How do you think the US Federal Government's yearly
         budget deficit has changed since January 2013?",
         "Increased a lot", "Increased a little ",
         "Stayed about the same", "Decreased a little ", "Decreased a lot"),
      Gender =person.Gender, Income =person.Income,
      Ethnicity =person. Ethnicity ,
};
query =from person in query where
      person.Income ==Income.INCOME_35_000_TO_49_999 &&
      person. Ethnicity ==Ethnicity.BLACK_OR_AFRICAN_AMERICAN
select person;
```

Figure 10: Budget deficit attitudes query.

3.3 Long-Tail Evaluation

A particularly poignant problem where yield optimizations help is queries for which the yield is very low, causing them to run for a long time to get a sufficient number of survey-takers who qualify. This is the so-called scurvy issue: if our qualifying question is whether the person suffers from scurvy, we will have to wait a very long time indeed to find enough qualifying participants.

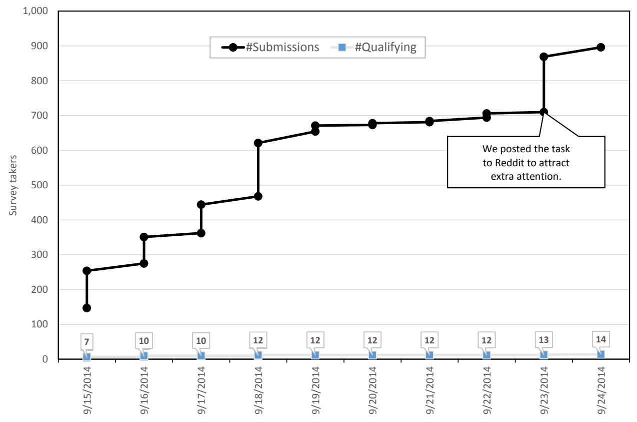


Figure 11: Completions for qualifications and passing for the budget query in Figure 10.

An more realistic example of such a query is given in Figure 10. We ran this query for 9 days. While about 900 people took the qualification test (black or African American with an income between \$35,000 and \$49,000), only 14 people in total qualified; out of these, the majority, 11 filled the rest of the poll asking about the budget deficit in the United States.

Figure 11 shows the number of survey takers which climbed steadily for four or five days before plateauing. To "boost" the interest in the task, we posted a link to our task on Reddit after about a week, which resulted in a surge of interest, with about 200 more people attempting to qualify, with two more actually qualifying.

We only paid .10¢ × 11 = \$1.1 for the entire survey, given that we do not pay unless the worker is qualified, as opposed to .10¢ × 900 = \$9. This is a difference of almost one order of magnitude in terms of the cost (8×). This longer-running query stresses our point about the importance of picking the right evaluation strategy.

4. Rebalancing

InterPoll supports answering decision questions of the form r_1 `boolOp` r_1, where both r_1 and r_2 are random variables obtained from segments of the population. Such a query is shown in Example 3, which compares the number of male and female liberal arts majors.

Question	Left filter	Right filter	Test
Do you believe that affirmative action on campus does more harm than good?	bachelors and more	some college	Left YES, Right NO
Do you believe that millennials don't stand a chance?	income 50K–75K ∨ income 75K+	income 25K–35K ∨ income 35K–50K	Left YES, Right NO
Do you believe that mass collection of U.S. phone records violates the fourth amendment?	age 35–44	age 18–24	Left YES, Right NO
Should we ration end-of-life care?	age <= 35	age > 35	Left YES, Right NO
Psychological poll about depression	income 50K–75K ∨ income 75K+	no income ∨ income 10K–15K ∨ income 15K–25K	Left less depressed than Right

Figure 12: Queries used for evaluating the rebalancing optimization summarized to save space.

To answer such *decision queries*, InterPoll repeatedly considers *pairs* of people from the categories on the left and right hand sides and then performs a sequential probability ratio test [22] to decide how many samples to request. A common problem, however, is that the two branches of the comparison are *unbalanced*: we are likely to have an unequal number of males and females in our samples, or people who are rich or poor, or people who own and do not own dogs.

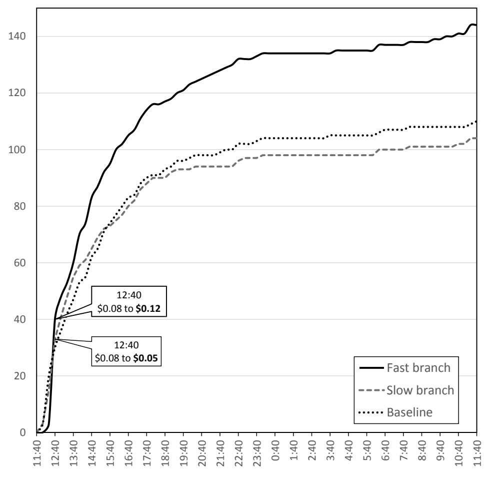
Lets assume that r_1 produces results faster than r_2. This is an issue because we if we end up paying for every person from r_1 and r_2 and we have too many from r_1, leading to us paying too much for samples we cannot use, and too few from r_2 creating a bottleneck in terms of speed with which we can create pairwise tests.

The focus of this subsection is on *re-balancing* the branches of such a test to even them out. The primary mechanism for re-balancing is changing the reward amount in an effort to attract more participants to the branch of the decision query that is too scarce. While many policies for changing the reward are possible, in practice, we tried increasing rewards by 50%. Starting with 8¢, we would bump up rewards for the slow branch up to 12¢ and then 15¢. We would also reduce the rewards for the slow branch to 5¢ and 3¢.

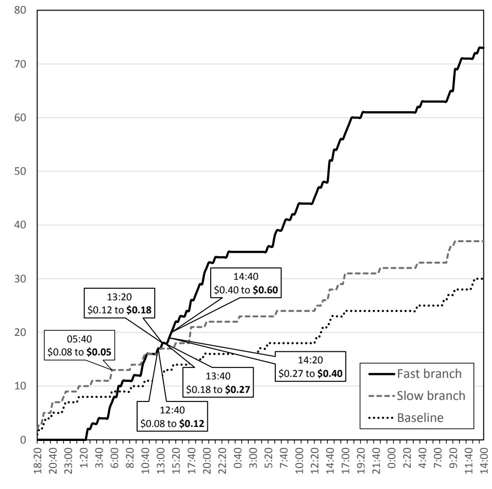
4.1 Evaluation

To evaluate the effectiveness of our rebalancing approach, we use the queries summarized in Figure 12. There are representative queries which all have a certain degree of disbalance between the fast and slow branch. To save space, we do not show the fully LINQ code for these five queries, only summarizing these queries by showing the question, left and right filters and the test (`binOp`).

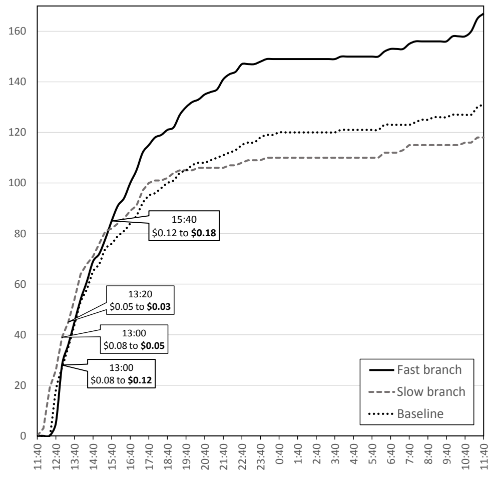
For comparison purposes, we also executed unoptimized versions of our queries, which we launch at the same times as the optimized ones, to reduce the effect of different times of day naturally attracting different numbers of people, for instance. Figure 13 shows the effect of the rebalancing optimization over time. Each chart compares the progress of the *fast branch*, the *slow branch*, as well as the default, unbalanced strategy over time. The x axis shows the time, in hours and minutes and the y axis shows the number



(a) Query for affirmative action.

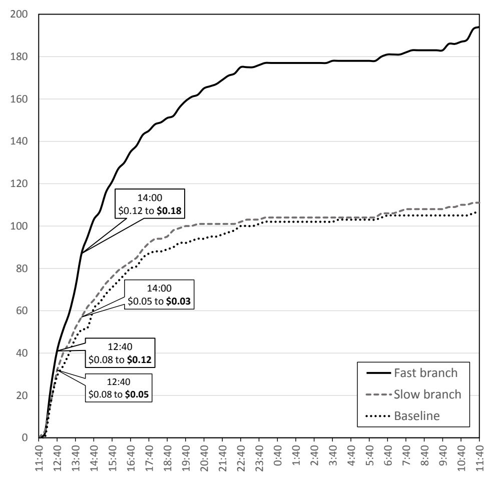


(b) Query for the millennials.

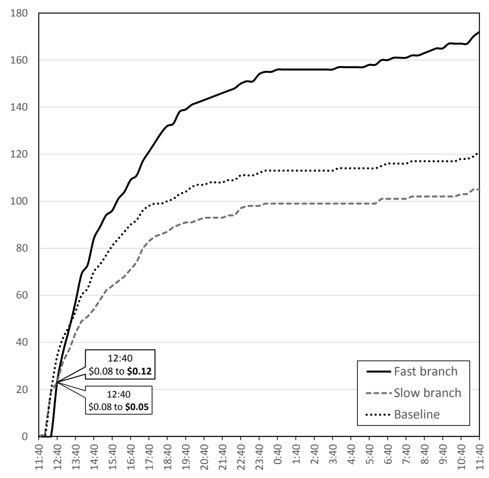


(c) Query for phone records privacy.

of completes (people) for each strategy. We also show the



(d) Query for rationing end-of-life care.



(e) Query for depression and anxiety.

Figure 13: Progress over time with and without rebalancing.

rebalancing points, i.e. points in time where we increase and decrease the rewards in call-outs on each chart.

In all cases, the solid black line (fast branch) attracts more people, at any point in time. In most cases, the slow branch and the default strategy both plateau pretty quickly.

In order to illustrate the time differences more explicitly, Figure 14 shows the effect of increasing the reward by measuring the time difference to get to x completes between the default strategy and the fast branch of our rebalancing strategy. While the number of completes (people) is shown on the x axis, the times are measured in hours, shown on the y axis. Overall, the time savings achieved through the use of rebalancing are significant: the fast branch gets to 70 completes over 2 hours faster and, for all strategies, to get to 90 completes, the it takes up to 21 more hours. Part of the explanation is in the fact that left to its own devices, a HIT on Mechanical Turk loses its popularity over time. One way to alleviate this problem is by bumping up the reward, to maintain an adequate level of interest.

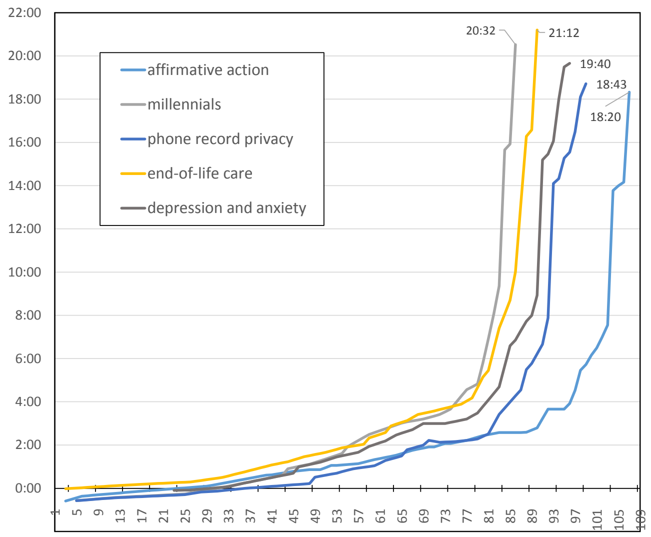
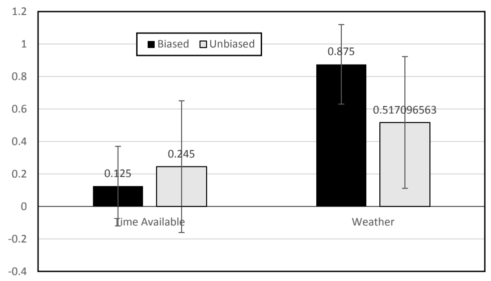
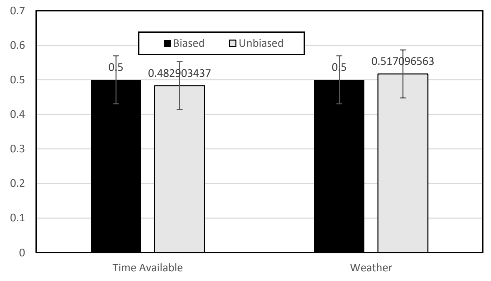


Figure 14: Time savings to get to x completes for the baseline approach vs. the fast branch. After about 50 people, the time differences become significant; after 70 people, they start exceeding several hours.



(a) Unrepresentative sample: 1 male and 7 females.



(b) Representative sample: 100 males and 100 females.

Figure 15: Unbiasing with different samples.

5. Panel Building

While InterPoll queries allow us to run opinion polls to measure sentiments of Mechanical Turk workers, most of the time, we want to measure opinions of people in the world at least, of at least those in a sufficiently big geographical area like the US. We start with a motivating example that illustrates the need for representative population samples.

Example 4 (Motivating example) Consider a somewhat

```
function $BuildPanel(D)$
Returns: panel $P$

 1: // Panel qualification mapping workers to categories
 2: $Q_P = CreateQualification()$
 3: // Qualification mapping workers to generation
 4: $Q_R = CreateQualification(autogrant = \texttt{true})$
 5: $U = FindUnrepresented(D, Q_P)$ // Unrepresented categories
 6: $G = 1$
 7: $Q_H = CreateQualTest(U, Q_R)$
 8: $H = CreateAndStartHit(Q_H)$ // HIT is created on mTurk
 9: while $U \neq \emptyset$ do
10:     $U' = U$
11:     $sleep(\texttt{1 hour})$
12:     $workers = H.GetWorkers()$
13:     // Move new workers to panel qualification
14:     $AssignQualification(workers, Q_P)$
15:     // mark them so that they don't retake subsequent hits
16:     $AssignQualification(workers, Q_R, G)$
17:     $U = FindUnrepresented(D, Q_P)$
18:     if $U \neq U' \wedge U \neq \emptyset$ then
19:         $Q_H.Dispose()$
20:         $H.Stop()$
21:         // Re-create qual-test and HIT for missing categories
22:         $G = G + 1$ // Next generation
23:         $Q_H = CreateQualTest(U, Q_R)$
24:         $H = CreateAndStartHit(Q_H)$
25:     end if
26: end while
27: $P = BuildPanel(Q_P)$
```

Figure 16: Algorithm BuildPanel for incremental panel creation.

artificial example that illustrates the difficulties of unbiasing with unrepresentative samples. Suppose we run the physical activity query in Figure 7c which asks people to comment on why they do not exercise more. Suppose men always blame the lack of time (choice `TimeAvailable`) and women blame the weather (`Weather`). If our sample consists of 1 man and 7 women, we attempt to unbias it to be representative of the population as captured by the Census. While the details of the process are outside the scope of this paper, here this involves providing unbiasing weights $0.48290343705769273/\frac{1}{8}$ for males and $0.51709656294230733/\frac{7}{8}$ for females.

Figure 15a shows both the biases and unbiased results, along with their 95% confidence intervals. We see that for both the `TimeAvailable` and `Weather` answers, the confidence intervals are really large, meaning the results are quite imprecise. Unbiasing only *increases* these intervals ($0.24 \to 0.40$).

However, in the case of an equal split of 100 men and 100 women (men still responding with `TimeAvailable` and women with `Weather`), the result in Figure 15b show very tight confidence intervals virtually unaffected by unbiasing ($0.06947 \to 0.06942$). $\square$

There are two takeaways of the example above: unbiasing unrepresentative samples can increase the confidence interval; there is no increase of the confidence interval for a representative sample. Also, having a representative sample obviates the need for unbiasing in the first place.

Avoiding unbiasing: It is therefore natural to try to avoid unbiasing as much as possible. Another most subtle reason is the ability to directly link the data that is obtained to individuals answering questions, thereby creating a "paper trail" for later use, following the philosophy of reproducible research. This is especially valuable if the researcher posting the query wants to ask some follow-up questions of particular segment of the responders.

Panel building: Consider the algorithm in Figure 16 which shows how to incrementally build the panel. One of the ways to achieve this is by pre-building a panel of workers whose demographic characteristics are pre-obtained and who demographic profile matches that of the general population. Doing so is no easy task. This is because there are some segments of the population that are genuinely tough to access.

For instance, people above the age of 75 are exceedingly difficult to find in the pool of Mechanical Turk workers. The same is true of people earning in excess of $100,000 per annum. Finding people who possess *both* of the above characteristics is that much more difficult. However, if somehow, over a long period of time, we are able to construct a panel and then reliably use the workers on that panels for running polls, the value of such a panel is tremendous. Arguably, the majority of value of companies such as Instant.ly[3] is in providing fast, on-demand preconstructed and responsive panels.

Our goal is to pre-build a panel and to use the panel to achieve better precision obviating the need to rely on unbiasing. This is similar to common practices of surveys in the physical world, where pollsters keep track of certain population segments that exhibit the required characteristics, and then they ask only those subsets that are of interest to take part in an upcoming survey.

Iterative panel construction: Algorithm BuildPanel in Figure 16 shows how a panel can be built iteratively, over a number of long-running *generations*. The goal of the algorithm is to create a panel P of people marked with a special qualification Q_P. Once the qualification Q_P has been granted to them, these people can be used on demand later on when we need to answer some queries. At every step of the algorithm, we query the currently running hit for new workers and then update U, the set of *unrepresented* categories. Function $FindUnrepresented(D)$ not shown here finds categories that are not adequately represented with respect to the provided target distribution D[4]. Whenever we reduce the set of unrepresented categories (check on line 18), we increment G, the generation counter and create a new hit to collect more samples. A dummy HIT is created with an increasingly narrow qualification test for missing categories. As time progresses, and more categories are completed, the current HIT is stopped and a new one is created with only the categories that are still incomplete.

5.1 Evaluation

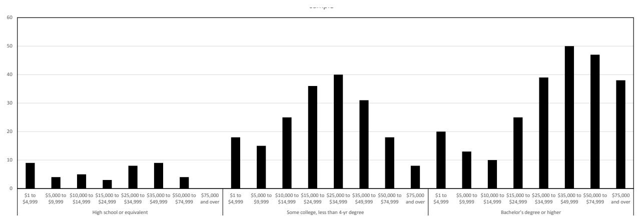
To test the ideas above, we decided to build a panel based on three demographic characteristics: *Gender*, *Education*, and *Income*. Considering (Male|Female) × (High School | Some college | Bachelors degree) × (8 levels of income) provides 48 categories to fill.

Let us emphasize that the process of panel building is likely to run for days if not months: at the time of submitting this paper, we have successfully filled 30 categories out of 48 in the following three dimensions. Figure 17 shows the histogram of our panel and the US Census[5]. To simplify the display, we only show 24 categories for each by combining

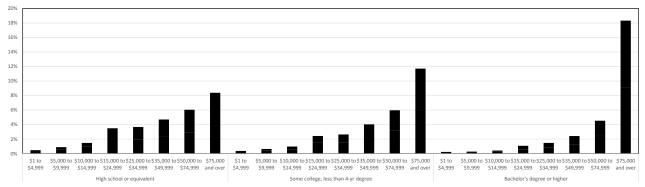
[3] `http.instant.ly`

[4] In our experiments, we typically use the US Census [32] as our target distribution. Unbiasing US-collected data using census statistics of other countries are another interesting possibility.

[5] We use absolute numbers for the former and percentages for the latter, but it is the shape of the histogram that matters.



(a) Our panel: breakdown across demographic categories.



(b) Census: breakdown across demographic categories.

Figure 17: Population histograms by category.

males and females. Note that the Mechanical Turk population is *relatively* balanced when it comes to the gender (for instance, 67%:43%, depending on the country). Focusing on other characteristics provides more of a stress test for our approach.

One can see that indeed, most categories are well-represented based on the shape of the histogram. Some categories, like less that $5,000 in income are pretty scarce on Mechanical Turk. Same is true of combinations like High school graduates who earn over $75,000 per annum. This is in part because people earning that much are less likely to appear on Mechanical Turk.

Example 5 (Height computation) The code below is an a query that asks workers to provide their height, in centimeters.

```
var frame =from person in people
     select new {
     Height =person.PoseQuestion<int>("What is your height?"),
     Gender =person.Gender,
      Ethnicity  =person. Ethnicity  };
var maleHeights =from person in frame
     where person.Gender ==Gender.MALE select person.Height;
var femaleHeights =from person in frame
     where person.Gender ==Gender.FEMALE select person.Height;
```

We limit our analysis to 50 respondents. Out of those, 35 were male and 15 female. The average height in the US, according to Wikipedia, is 176.3 cm for males and 162.2 cm for females. Assuming a 52%:48% mix in the population, the average height is 169.4.

Below, we report mean and 95% confidence intervals for the different mixes. The equal 15:15 mix places the true value of 169.4 well within the confidence interval, close to the mean. The very unequal 1:35 mix is a bad predictor of the true value: 169.4 is *outside* the confidence interval. We can see that the unbiasing based on the very unequal 1:35 mix also places the real value of 169.4 outside the confidence interval.

This example highlights the importance of having a balanced sample. A more drastic example is unbiasing based on ethnicity. Considering the female sub-population only, the mean height is *virtually the same* as the Wikipedia value. Unbiasing the female heights with respect to ethnicity, however, produces 166.15 ± 5.32 cm. This is significantly larger than

Sub-population	**Mean**	**±CI**	**Wikipedia**	**Within**
Male	175.1	4.58	176.3	✓
Female	162.9	4.89	162.2	✓
Equal mix (15:15)	168.1	4.20	169.4	✓
More balanced mix (35:15)	171.48	3.83	169.4	✓
More balanced mix unbiased	168.82	3.68	169.4	✓
Unequal mix (35:1)	174.5	4.63	169.4	✓
Unequal mix unbiased	163.1	5.73	169.4	✗

the true value and the difference emerges when we discover that our sample is reweighed using a weight of 50 for African Americans (12% in the population but only 1 in our sample). This taller woman (188 cm) has a large effect on the unbiased mean. ☐

6. Related Work

Perhaps the most closely-related recent project is SurveyMan by Tosch *et al.* [30]; SurveyMan provides a lightweight-specific language to survey developers. In InterPoll, we choose to use built-in language-integrated queries, which are already familiar to many developers. The chief focus of Tosch is to ensure survey and result quality. SurveyMan statically analyzes surveys to provide feedback to survey authors before deployment. SurveyMan's dynamic analyses aim to automatically find survey bugs, and control for the quality of responses. We consider SurveyMan to be complimentary to the work described in this paper: indeed, many of the techniques presented by Tosch *et al.* can be employed to improve InterPoll queries and response quality; however, the focus of this particular paper is on optimizations.

We do not aim to adequately survey the vast quantity of crowd-sourcing-related research out there; the interested reader may consult [34]. A great deal of work has focused on matching users with tasks, quality control, decreasing the task latency, etc. Moreover, we should note that our focus is on subjective *opinion polls*, which distinguishes InterPoll work from the majority of crowd-sourcing research, which requires "solving" a particular problem, i.e. deciphering a license plate number in a picture, translating sentences, etc. In InterPoll, we are primarily interested in self-reported opinions of users about themselves and the world.

6.1 Crowd-Sourcing Systems

There has been a great deal of interest in recent years in building new systems for automating crowd-sourcing tasks. TurKit [20] is one of the first attempts to automate programming crowd-sourced systems. The developer can write TurkIt scripts using JavaScript. AutoMan [2] is a programmable approach to combining crowd-based and regular programming tasks, a goal shared with Truong *et al.* [31]. The focus of AutoMan is on computation reliability, consistency and accuracy of obtained results, as well as task scheduling. Turkomatic [18, 19] is a system for expression crowd-sourced tasks and designing workflows. CrowdForge is a general purpose framework for accomplishing complex and interdependent tasks using micro-task markets [17]. Some of the tasks involve article writing, decision making, and science journalism, which demonstrates the benefits and limitations of the chosen approach. CrowdWeaver is a system to visually manage complex crowd work [16]. The system supports the creation and reuse of crowd-sourcing and computational tasks into integrated task flows, manages the flow of data between tasks, etc. Quizz [14] is a gamified crowdsourcing system that simultaneously assesses the knowledge of users and acquires new knowledge from them.

6.2 Optimizing Crowd Queries

CrowdDB [10] uses human input via crowd-sourcing to process queries that regular database systems cannot adequately answer. For example, when information for IBM is missing in the underlying database, crowd workers can quickly look it up and return as part of query results, as requested. CrowdDB uses SQL both as a language for posing complex queries and as a way to model data. While CrowdDB leverages many aspects of traditional database systems, there are also important differences. Qurk implements a number of optimizations [24], including task batching, replacing pairwise comparisons with numerical ratings, and pre-filtering tables before joining them, which dramatically reduces the overall cost of sorts and joins on the crowd. End-to-end experiments show cost reductions of 14.5× on tasks that involve matching up photographs and ordering geometric pictures. These optimization gains in part inspire our focus on cost-oriented optimizations in InterPoll. Marcus *et al.* [23] study how to estimate the *selectivity* of a predicate with help from the crowd, such as filters photos of people to those of males with red hair.

6.3 Database and LINQ Optimizations

A survey of query optimizations in databases, with a focus on join optimizations, among others is presented in Ioannidis [13]. While language-integrated queries are wonderful for bringing the power of data access to ordinary developers, LINQ queries frequently do not result in most efficient executions. There has also been interest in both formalizing the semantics of [5] and optimizing LINQ queries. Grust *et al.* propose a technique for alternative efficient LINQ-to-SQL:1999 compilation [11]. Steno [26] proposes a strategy for removing some of the inefficiency in built-in LINQ compilation and eliminates it by fusing queries and iterators together and directly compiling LINQ queries to .NET code. Nerella *et al.* [27] relies on programmer-provided annotations to devise better queries plans for language-integrated queries in JQL, Java Query Language. Annotations can provide information about shapes of distribution for continuous data, for example. Schueller *et al.* [28] focus on bringing the idea of *update propagation* to LINQ queries and combining it with reactive programming. Tawalare *et al.* [29] explore another compile-time optimization approach for JQL. Bleja *et al.* [3] propose a new static optimization method for object-oriented queries dealing with a special class of sub-queries of a given query called “weakly dependent sub-queries”.

7. Conclusions

This paper develops a range of both static and dynamic optimizations for crowd-sourced queries, which can be used to combine human and machine computation. Unlike a typical optimization effort, one of the clear upsides of our work is that not only it saves time for the user, it also saves *money*, often *hundreds* and *thousands* of dollars, when queries are run at scale.

Optimizations in this paper lead to significant improvements in practice. In our experiments we observed *tenfold* savings in survey cost and time savings of as much as 20 hours for some of the longer-running queries. Our work translates language and runtime optimization ideas into real and measurable financial savings. Relying on automatically pre-built penels can also increase result precision and decrease the confidence intervals.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1988.

[2] D. Barowy, C. Curtsinger, E. Berger, and A. McGregor. AutoMan: A platform for integrating human-based and digital computation. *Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA ’12*, page 639, Jan. 2012.

[3] M. Bleja, T. Kowalski, and K. Subieta. Optimization of object-oriented queries through rewriting compound weakly dependent subqueries. *Database and Expert Systems*, pages 1–8, Jan. 2010.

[4] J. Bornholt, T. Mytkowicz, and K. S. Mckinley. Uncertain<T>: A first-order type for uncertain data. 2013.

[5] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming - ICFP ’13*, page 403, Jan. 2013.

[6] C. Cooper, D. M. McCord, and A. Socha. Evaluating the college sophomore problem: the case of personality and politics. *The Journal of psychology*, 145(1):23–37, 2011.

[7] M. P. Couper. Review: Web surveys: A review of issues and approaches. *The Public Opinion Quarterly*, pages 1–31, Jan. 2000.

[8] D. Dillman, R. Tortora, and D. Bowker. Principles for constructing Web surveys. 1998.

[9] J. Evans, N. Hempstead, and A. Mathur. The value of online surveys. *Internet Research*, 15(2):195–219, Jan. 2005.

[10] M. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: answering queries with crowdsourcing. *SIGMOD ’11: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1–12, June 2011.

[11] T. Grust, J. Rittinger, and T. Schreiber. Avalanche-safe LINQ compilation. *Proceedings of the VLDB Endowment*, 3(1-2):162–172, Sept. 2010.

[12] H. Gunn. Web-based surveys: Changing the survey process. *First Monday*, 2002.

[13] Y. E. Ioannidis. Query optimization. *ACM Computer Surveys*, 28(1), Mar. 1996.

[14] P. Ipeirotis and E. Gabrilovich. Quizz: Targeted crowdsourcing with a billion (potential) users. In *WWW*, 2014.

[15] S. Keeter, L. Christian, and S. Researcher. A Comparison of Results from Surveys by the Pew Research Center and Google Consumer Surveys. 2012.

[16] A. Kittur, S. Khamkar, P. André, and R. Kraut. CrowdWeaver: Visually Managing Complex Crowd Work. *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work - CSCW ’12*, page 1033, Jan. 2012.

[17] R. E. Kraut. CrowdForge : Crowdsourcing Complex Work. *UIST*, pages 43–52, 2011.

[18] A. Kulkarni, M. Can, and B. Hartmann. Collaboratively crowdsourcing workflows with turkomatic. *of the ACM 2012 conference on*, Jan. 2012.

[19] A. P. Kulkarni, M. Can, and B. Hartmann. Turkomatic: automatic recursive task and workflow design for mechanical turk. *CHI’11 Extended Abstracts on Human*, Jan. 2011.

[20] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. TurKit: tools for iterative tasks on Mechanical Turk.

Proceedings of UIST, pages 1–2, Jan. 2009.

[21] B. Livshits and T. Mytkowicz. Interpoll: Crowd-sourced internet polls (done right). Technical Report MSR-TR-2014-3, Microsoft Research, Jan. 2014.

[22] B. Livshits and T. Mytkowicz. Saving money while polling with InterPoll using power analysis. In *Conference on Human Computation and Crowdsourcing*, 2014.

[23] A. Marcus, D. Karger, S. Madden, R. Miller, and S. Oh. Counting with the crowd. *Proceedings of the VLDB Endowment ,*, 6(2), Dec. 2012.

[24] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Human-powered sorts and joins. *Proceedings of the VLDB Endowment ,*, 5(1), Sept. 2011.

[25] J. Mayo. *LINQ Programming*. McGraw-Hill Osborne Media, 1 edition, 2008.

[26] D. Murray, M. Isard, and Y. Yu. Steno: automatic optimization of declarative queries. *Proceedings of the Conference on Programming Language Design and Implementation*, pages 1–11, June 2011.

[27] V. Nerella, S. Madria, and T. Weigert. An Approach for Optimization of Object Queries on Collections Using Annotations. *2013 17th European Conference on Software Maintenance and Reengineering*, pages 273–282, Mar. 2013.

[28] G. Schueller and A. Behrend. Stream fusion using reactive programming, LINQ and magic updates. *Proceedings of the International Conference on Information Fusion*, pages 1–8, Jan. 2013.

[29] S. Tawalare and S. Dhande. Query Optimization to Improve Performance of the Code Execution. *Computer Engineering and Intelligent Systems*, 3(1):44–52, Jan. 2012.

[30] E. Tosch and E. D. Berger. Surveyman: Programming and automatically debugging surveys. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2014.

[31] H. L. Truong, S. Dustdar, and K. Bhattacharya. Programming hybrid services in the cloud. *Service-Oriented Computing*, pages 1–15, Jan. 2012.

[32] US Census. Current population survey, October 2010, school enrollment and Internet use supplement file. (October), 2010.

[33] J. Wyatt. When to use web-based surveys. *Journal of the American Medical Informatics Association*, 2000.

[34] X. Yin, W. Liu, Y. Wang, C. Yang, and L. Lu. What? How? Where? A Survey of Crowdsourcing. *Frontier and Future Development of*, Jan. 2014.