ChatGPT

Ibis Python Framework Reference (v5+)

Overview and Deferred Execution Model

Deferred Evaluation: Ibis represents data transformations as **expressions** without immediate execution. By default, expressions are built into a query plan and only executed when needed (e.g. when calling `execute()` or converting to a pandas DataFrame). In **deferred mode**, printing an expression shows its internal tree (like an AST) rather than actual data [1]. For example:

```
t = ibis.examples.penguins.fetch()        # Lazy table expression
print(t.head(3))
```

This might display an internal representation (with an alias like `r0 := DatabaseTable: penguins` and a `Limit[r0, n=3]` operation) instead of the data [2] [3]. In deferred mode, no query runs until explicitly executed.

Interactive Mode: For convenience in exploratory use, you can enable *interactive mode* to auto-execute on print. Setting `ibis.options.interactive = True` causes any table expression printed in a REPL to trigger execution and show results immediately [4]. This mode is optional; under the hood Ibis still constructs a deferred query but fetches the result for display. For production code and large data, deferred (lazy) execution is typically preferred.

Expression Objects: All Ibis computations are represented by expression objects forming a **query graph**. Key expression types include: - **Table expressions:** Represent tabular data (analogous to a DataFrame or SQL table) with named columns. A table expression has a schema of columns with fixed data types. Table expressions can come from database tables, file reads, in-memory data, or even *unbound tables* defined purely by schema. - **Column expressions:** Represent a single column of a table (or an array of values). These are usually produced by selecting a column from a table (e.g. `t.column_name`) or as the result of an expression applied to a column. In Ibis, column expressions are a subtype of **value expressions** with an array (vector) semantics – they yield one value per row when evaluated. - **Scalar expressions:** Represent single values (0-dimensional). These can arise from aggregations (e.g. `t.column.mean()` produces a scalar expression) or literal values. Scalar expressions can be standalone or appear in computations (e.g. adding a scalar to a column yields a column expression). - **Literals:** Ibis provides `ibis.literal(value, type?)` to embed Python literal values as Ibis expressions. For example, `ibis.literal(42)` is a scalar expression representing the constant 42 (type int64 by default), and `ibis.literal(["a", "b"], type="array<string>")` could represent an array literal. Literal expressions are useful for comparing against columns or as UDF inputs, and they carry type information in the Ibis type system.

Unbound Tables: You can define a table by schema alone, without initial data or backend. For example, `schema = {"id": "int64", "value": "string"}; t = ibis.table(schema, name="mytable")` creates an *unbound* table expression. This acts like an empty table with known column types – “an empty spreadsheet with just the header" [5]. You can perform transforms on it (filter, group, etc.) as you would on real data. The actual data can be bound later for execution. Unbound tables let you write backend-agnostic transformations in advance. When ready, you can execute the expression by connecting to a backend and binding the expression to real data (e.g., via `con.execute(expr)` or by inserting data and calling `expr` on that connection) [6] [7].

Chaining with Deferred Expressions: Ibis supports a concise **deferred execution API** using an underscore placeholder. By importing `from ibis import _`, you can write expressions without naming the table explicitly each time. The underscore `_` represents the *current table* in a chain of expressions [8]. For example:

```
from ibis import _, selectors as s
result = (
    t
    .filter(_.year > 2008)
    .mutate(ratio = _.colA / _.colB)
    .group_by(_.category)
    .agg(total=_.ratio.sum(), avg=_.value.mean())
    .filter(_.total > 100)
)
```

In this chain, `_.year` refers to `t.year` within the filter, and later `_.total` refers to the aggregated total in the second filter. The underscore makes it easy to **chain transformations** in one fluent expression [9] [10]. It also allows defining reusable column computations: e.g. `high_variance = (_ - _.mean())/_.std()` can be applied across many columns via selectors (see **Column Selectors** below). Note that `_` can only be used inside an expression chain or in functions that accept deferred expressions; it resolves to the appropriate table or value in context.

Table Expressions and Core Operations

A **table expression** in Ibis has a set of named columns with fixed types (see **Schema and Types** below). Table expressions support relational algebra operations similar to SQL and pandas. Unless otherwise noted, these operations **do not execute immediately** but produce new deferred expressions. Multiple operations can be composed into a single query.

Projection and Column Selection

To select or project columns, use `table.select(...)`. This can rename or compute new columns as well. For convenience, you can use `table[columns]` or `table[[...]]` for simple column subset selection (similar to pandas/DataFrame indexing). For example, `t[["col1", "col3"]]` returns a new table expression with only those two columns.

You can also add or replace columns using `mutate()`. `table.mutate(new_col = expr, ...)` returns a table expression with additional or updated columns [11] [12]. For instance:

```
t2 = t.select("id", "value").mutate(value_len = t.value.length())
```

This projects only `id` and `value` from `t`, then adds a computed column `value_len`. (Under the hood `mutate` is similar to `SELECT ..., <expr> as value_len` in SQL.)

Row Filtering (WHERE clause)

Use `table.filter(predicate)` to filter rows by some condition (predicate expression). The predicate is an expression yielding boolean (True/False) per row. For example, `t.filter(t.year == 2009)` yields a table with only rows where year equals 2009. Multiple conditions can be combined via bitwise operators (`&` for AND, `|` for OR, `~` for NOT). E.g. `t.filter((t.value > 0) & (t.flag == 'Y'))`.

Deferred predicate example: Using the underscore API, one can write `t.filter(_.colA.contains("foo") | (_.colB == 0))` where `_` refers to `t` inside the filter. Filtering does not run the query; it returns a new filtered table expression.

Sorting and Limiting

Use `table.order_by([keys...])` to sort. For descending order, wrap a column in `ibis.desc()` (or `ibis.asc()` for explicit ascending) [13]. For example: `t.order_by([ibis.desc(t.total)])`. Sorting returns a new table expr with an ordering (backends will include an `ORDER BY` in the SQL).

Use `table.limit(n, offset=0)` to restrict the number of rows (like SQL `LIMIT`). For example, `t.limit(100)` will cap results to 100 rows [14] [15]. (Note: if no explicit order, the subset of rows is arbitrary; apply `order_by` for deterministic results [16].) You can also slice using Python indexing on the table expression, e.g. `t[:100]` or `t[100:200]` – this is equivalent to limit/offset. By default, Ibis applies a limit of 10,000 rows on execution to prevent accidentally fetching huge data (configurable via `ibis.options.sql.default_limit`) [17].

Aggregation and Group By

Grouping: Use `table.group_by(keys).agg(**metrics)` to perform aggregations. This is analogous to SQL `GROUP BY` or pandas `groupby().agg()`. The `keys` can be one or more column names or expressions to group on. The `metrics` are defined as keyword arguments: each is an aggregate expression applied per group. Common aggregates include `.count()`, `.sum()`, `.mean()`, `.max()`, `.min()`, `.median()`, `.std()` (standard deviation), `.nunique()` (count distinct), etc., as methods on column expressions. For example:

```
# aggregate total sales and average price by store and category
result = sales.group_by(["store_id", "category"]).agg(
    total_sales = sales.amount.sum(),
    avg_price   = sales.unit_price.mean()
)
```

This yields a new **table expression** with columns: store_id, category, total_sales, avg_price. Under the hood, Ibis will compile a SQL query with GROUP BY store_id, category and the specified aggregates.

Scalar vs Table Aggregate: If you call an aggregate function on a table *without* grouping, it produces a **scalar expression** – e.g. `t.column.sum()` returns a scalar (the sum of entire column). If you want that as a Python value, you need to execute it (e.g. `con.execute(t.column.sum())`). If you use `agg` with no group_by, it returns a **single-row, single-column table** containing the result (which when executed yields one row). In practice, `table.aggregate(metric=...)` is similar to `SELECT ... FROM table` with no GROUP BY (whole table aggregation).

Having filters: Ibis allows filters on aggregates via `.filter()` on an aggregated table expression. For example, you might do `agg = sales.group_by("store").agg(total=_.amount.sum()); agg.filter(_.total > 1000)`. This corresponds to a SQL query with a `HAVING total > 1000`.

Joins (Equijoins, Non-Equi Joins, Lateral)

Ibis supports a variety of SQL join types through the `join` method or specialized join functions: - **Equi-joins:** The most common case – join on one or more key columns equal on both sides. Use `table1.join(table2, predicates, how="inner")`. By default `how='inner'` (inner join) [18], but you can specify `'left'`, `'right'`, `'outer'` (full outer), `'semi'`, `'anti'` as needed. The `predicates` can be: a column name (string) for joining on columns with the same name, a list of column names, or a boolean expression (e.g. `table1.col1 == table2.col2`). For example, `orders.join(customers, "customer_id", how="left")` joins on `customer_id` with a left-join [19]. Multiple keys example: `t1.join(t2, ["colA", "colB"])` joins on both colA and colB matching [20]. You can also pass a list of expressions: e.g. `[t1.x == t2.y, t1.category == t2.category]`.

Join output: The result is a new table expression. By default, if there are overlapping column names, Ibis will suffix them with `_right` (for right table) or use provided `lname`/`rname` templates [21]. You can then `.select()` the desired fields, possibly renaming. Alternatively, instead of `join` then `select`, you can use convenience methods like `t1.inner_join(t2, ...).select(...)` or directly `t1.join(t2, ..., how="inner").select(...)` to project specific fields from both tables [22].

- **Non-equi joins:** Ibis also allows join conditions other than equality. The join predicate can be any boolean expression combining columns of both tables. E.g. `t1.join(t2, t1.value >= t2.threshold, how="inner")` is valid if the backend supports it. Some analytical use-cases (like band join, ranges) can be expressed this way. Note that not all backends support arbitrary non-equi joins – SQL engines often have limitations. Check the backend's docs or operation support matrix if using non-standard join conditions.

- **As-of joins:** Ibis provides `asof_join` for time-series or nearest-key joins. An *as-of join* is like a left-join that matches on keys and "closest preceding" criteria. For example, `t1.asof_join(t2, on="timestamp", by="id", tolerance=interval)` will join each row of t1 to the most recent row in t2 (by timestamp not greater than t1's timestamp) sharing the same id, within some time `tolerance` [23] [24]. This is useful for things like joining a table of events with a table of regularly sampled measurements. The `on` parameter is the timestamp (or ordered key) for as-of matching, and `by`/`predicates` are equality keys to match exactly. The joined result will have columns of both tables (with suffixes as needed) – e.g. columns from `right` might be suffixed with `_right` if names conflict [25].

- **Cross joins:** Use `table1.cross_join(table2)` for Cartesian product (all combinations of rows) [26]. This is equivalent to SQL `FROM t1 CROSS JOIN t2`. It can also take multiple tables: `t1.cross_join(t2, t3, ...)`.

- **Semi/Anti joins:** Ibis supports semi joins (return rows from left that have a match in right) and anti joins (return rows from left that *do not* have a match). These are typically methods `table1.semi_join(table2, predicates)` and `anti_join`. Semantically, a semi join is like an `EXISTS` subquery, returning only left table columns, and an anti join is like `LEFT ANTI` (or `NOT EXISTS`). Under the hood, Ibis compiles these appropriately if the backend supports them. (Not all backends have direct support; Ibis may emulate via subqueries.)

- **Self Joins:** To join a table with itself, use the `.view()` method to create an independent reference to the same table [27]. For example:

```
t1 = table.view()
result = table.join(t1, table.col == t1.col, how="inner")
```

  Here `table.view()` gives a duplicate of `table` that Ibis treats as a separate relation (often assigning it an alias). This avoids confusion of column references in the join condition. Use this for self-joins or when you need the same table multiple times in one query.

- **Complex join conditions (Lateral joins):** The join predicate in Ibis can be a tuple of expressions for more advanced scenarios [28] [29]. For example, you can join on a condition that a transformed value from one table equals a transformed value from another. The predicate tuple form `(left_expr, right_expr)` means "evaluate these two expressions and join where they are equal." Each side can be a deferred expression or even a lambda. A practical example: to find pairs of movies that share the same tag (ignoring case), one can do:

```
movie_tags = tags[["movieId", "tag"]]
mt2 = movie_tags.view()
joined = movie_tags.join(mt2, [
    movie_tags.movieId != mt2.movieId,                       # different
movie
    (movie_tags.tag.lower(), lambda t: t.tag.lower())         # same tag
lowercased
])
```

  This will perform a self-join of `movie_tags` with itself (aliased as `mt2`), with two conditions: movieIds not equal (so we don't join a movie to itself), and the tag strings match in lowercase [29] [30]. The result will have columns `movieId`, `tag`, `movieId_right`, `tag_right` (the second being the matching movie). This pattern effectively does a lateral self-join on a computed key.

Lateral joins / Unnest: Some databases support *lateral joins* (e.g. Postgres `JOIN LATERAL`) for subqueries or for array unnesting. In Ibis, *unnesting* an array or map column is supported via `table.unnest()`. For an array column `arr`, `table.unnest("arr", offset="idx")` will produce a new table where each element of the array becomes a separate row (duplicating the other columns), with an optional generated sequence column `idx` for the element index [31] [32]. This compiles to a backend-specific lateral join or explode. The `keep_empty=True` option will keep original rows that have empty/ NULL arrays (with NULL for the unnested value) [33]. Unnesting is often used with JSON or array data that needs to be flattened. The resulting table can be further joined with others if needed (conceptually similar to SQL CROSS JOIN LATERAL).

Set Operations (Union/Intersect/Except)

Ibis supports set algebra on tables: - **Union:** `t1.union(t2)` returns a new table expression that is the union of rows from `t1` and `t2`. By default this is a multiset union (duplicates retained, i.e. SQL "UNION ALL") [34]. To get distinct union (like SQL `UNION`), pass `distinct=True`: `t1.union(t2, distinct=True)` [35]. You can union more than two tables by chaining or passing multiple operands (`t1.union(t2, t3, ...)`). All tables must have the same schema.
- **Intersection:** `t1.intersect(t2)` gives the intersection of two tables (rows present in both). This typically yields distinct rows common to both. It may not be supported on all backends; some might emulate it via joins or subqueries. - **Difference/Except:** `t1.difference(t2)` (or sometimes `.except_` in Ibis versions) gives rows in `t1` that are not in `t2` (SQL EXCEPT or MINUS). Again, support may vary by backend.

These operations are useful for set-based comparisons or combining results from multiple queries. Under the hood, Ibis composes the SQL set operation if available (e.g., using `UNION ALL` or `UNION DISTINCT` etc.) or falls back to an alternative strategy.

Window Functions and Analytics

Ibis provides robust support for window (analytic) functions, which allow computations across sets of rows relative to the current row (similar to SQL window functions using `OVER` clause).

Defining Windows: Use `ibis.window(...)` to define a window. Common parameters: - `group_by` – grouping keys for partitioning (like `PARTITION BY` in SQL). - `order_by` – sorting within partition (like `ORDER BY` in window). - `preceding` and `following` – define a frame relative to the current row (e.g. how many rows before/after to include). You can use an integer number of rows or `None` for unbounded. For example, `ibis.window(order_by=col, preceding=2, following=0)` defines a window of the current row and 2 preceding rows. - Alternatively, you can use `range_window` for time/date ranges (if supported by backend).

Using `.over()`: Given a window object `win`, you can apply it to an analytic function by calling `.over(win)`. Many aggregate or analytic functions in Ibis can act as window functions when followed by `.over(window)`. For example:

```
win = ibis.window(group_by="category", order_by="timestamp", preceding=None,
following=0)
```

```
t = t.mutate(
    row_num = ibis.row_number().over(win),
    cum_sum = _.value.sum().over(win)
)
```

Here `ibis.row_number().over(win)` gives a sequential row number per category partition [36], and `_.value.sum().over(win)` gives a running cumulative sum of `value` within each category, ordered by timestamp. These new columns (`row_num`, `cum_sum`) are added to the table expression. When executed on a backend like PostgreSQL or DuckDB, they compile to `ROW_NUMBER() OVER(PARTITION BY category ORDER BY timestamp)` and `SUM(value) OVER(PARTITION BY category ORDER BY timestamp ROWS UNBOUNDED PRECEDING)` respectively.

Ranking and Lag/Lead: Ibis has built-in analytic functions such as `ibis.row_number()`, `ibis.rank()`, `ibis.dense_rank()`, `ibis.percent_rank()`, etc. These return a value expression that must be bound to a window with `.over()`. For example, `ibis.rank().over(ibis.window(order_by=_.score.desc()))` would rank rows by score. In addition, column expressions have methods `lag(n=1, default=None)` and `lead(n=1, default=None)` to access preceding or following values in a window. For example, `t.mutate(prev_val = t.value.lag())` will, when executed with an appropriate window ordering, give the previous row's value (similar to `LAG(value) OVER(...)` in SQL) [37]. By default, `lag()` and `lead()` require an implicit sort order – it is recommended to always use them with an explicit window specification via `.over()` to avoid non-determinism.

Example: Forward fill (carry last observation forward) using window functions:

```
win = ibis.window(order_by=table.timestamp, following=0)  # all rows up to
current row
grouped = table.mutate(grouper = table.value.count().over(win))
result = grouped.group_by(grouped.grouper).mutate(
    ffill = grouped.value.max()    # max within each group (the single non-null)
)
```

In this pattern, we created a window `win` ordering by time, then defined `grouper` as a rolling count of non-nulls over that window [38]. This `grouper` essentially partitions the data into segments where only one non-null appears. Then grouping by this `grouper` and taking the max yields forward-filled values [39]. (This is an idiom for forward-fill; backward fill would use a window with reverse order [13].)

Notes: Not all backends implement every window function. Simple cases like row_number and partitioned aggregates are widely supported. More complex frames (e.g. rows between specific offsets, or range between values) may not be available on all engines. Ibis will throw an `UnsupportedOperationError` at compile time if the backend cannot handle a given window specification. Always test critical window logic on your target backend.

Advanced Data Types and Operations

Ibis can handle **complex data types** beyond flat relational data, including *structs*, *arrays*, *maps*, and *JSON*. These manifest as expression types with special methods for nested access and manipulation.

Structs (Nested Structures)

A **Struct** in Ibis is a compound data type with named fields (like a JSON object or a C struct). For example, a struct value might have fields `a: int64` and `b: string`. In Ibis, a struct is represented as a `StructValue` expression. You can construct a struct literal using `ibis.struct({...})` or by creating a Python dict and casting. Struct columns can also come from backend data (e.g. a table column of SQL type STRUCT).

Accessing Fields: You can access struct fields by name using **dot notation** or dictionary-style indexing. For a table `t` with a struct column `s`, `t.s.fieldname` will yield the value of the `fieldname` field as a new column expression [40]. Equivalently, `t.s["fieldname"]` does the same [40]. These operations produce an expression of the field's type (e.g. int64). For example: if `t.s` is `struct<a: int64, b: string>`, then `t.s.a` is an `int64` column expression [41]. If a struct field is itself a complex type, you can chain access (e.g. `t.s.substruct.innerfield`). Ibis ensures both `.` and `[]` access are supported and compiles to the appropriate backend syntax (`s->'a'` or `s.a` etc., depending on SQL dialect).

Constructing Structs: Use `ibis.struct({"f1": expr1, "f2": expr2, ...})` to create a struct expression from individual field expressions. This is useful to bundle multiple columns into one struct (some backends support struct construction, others might fall back to JSON). For example, `ibis.struct({"a": t.col1, "b": t.col2})` yields a struct expression with fields a and b. You can assign this to a new column via `mutate(combined=ibis.struct({...}))`. You can also create a literal struct from Python data: e.g. `ibis.literal({"a": 5, "b": "foo"})` will infer a struct type `<a: int64, b: string>`.

Arrays (Lists)

An **Array** is a sequence of items of a uniform type (analogous to a list in Python or a SQL array). In Ibis, array columns are `ArrayValue` expressions, and you can have array literals as well.

Constructing Arrays: Use `ibis.array([val1, val2, ...])` to create a literal array expression. If the Python list is heterogeneous or cannot infer type, you may need to specify the element type. If you have columns and want to form an array, some backends support functions to collect aggregations into arrays (e.g. BigQuery's ARRAY_AGG); Ibis provides high-level operations but direct array construction from multiple columns might not be portable.

Array Methods: Ibis offers many methods to work with arrays: - **Element selection:** If `arr` is an array expression, `arr[i]` (using `[]` with an integer expression) gives the element at index `i` (0-based). E.g. `t.arr_col[0]` is the first element of each array. (Some backends use 1-based indexing; Ibis handles translation.) - **Length:** `arr.length()` returns the length of each array (number of elements) [42]. - **Contains:** `arr.contains(item)` yields a boolean expression checking if `item` is present in the array. - **Slice/Subarray:** `arr[start:end]` slicing is supported in some contexts to get a subarray. - **Set-like ops:** `arr.union(other_array)`, `arr.intersect(other_array)`, `arr.unique()` for distinct elements, etc., are provided to perform element-wise set operations on arrays [43] [44]. For example, `t.tags.union(t.more_tags)` might produce an array of unique elements from the combination of two array columns element-wise (if backend supports array union). - **Flatten:** `arr.flatten()` will concatenate a nested array of arrays into a single flat array [45] (one level of flattening). - **Map/apply:** You can apply a scalar function to each element using `arr.map(lambda x: <expression using x>)`. For instance, `arr.map(lambda x: x + 1)` adds 1 to each numeric element [46]. This is subject to backend support (not all SQL dialects allow arbitrary lambda on arrays; some may compile to UNNEST + SELECT). - **Zip:** `arr1.zip(arr2, ...)` merges multiple arrays elementwise into an array of structs (pairing elements by index) [47]. - **Unnesting:** As mentioned earlier, `table.unnest("array_col")` will turn an array column into multiple rows (useful to analyze array contents at the top level).

Example: Suppose `t.features` is an array<int> column. You can do: `t.select(t.id, feature_count = t.features.length())` to get the count of features per row. To filter rows where a certain feature appears: `t.filter(t.features.contains(42))`. Or to extract the first feature: `t.features[0]`. If you want to explode the array into long format: `t.unnest("features", offset="idx")` will produce a table with columns `features` (each element as a separate row) and `idx` (the original position) [32], along with any other original columns.

Maps (Key-Value Dictionaries)

A **Map** in Ibis holds key-value pairs (associative array). Keys and values each have a specific data type (all keys same type, all values same type). Maps in Ibis are often backed by SQL maps (e.g. in DuckDB or BigQuery) or Python dicts in the pandas backend.

Constructing Maps: Use `ibis.map({key1: val1, key2: val2, ...})` to create a map literal. For example:

```
m = ibis.map({"a": 1, "b": 2})
```

This yields a `MapValue` expression with type `map<string, int64>` (if types can be inferred) [48]. Map columns can also come from reading from a backend table that has a map type.

Map Methods: - **Retrieve by key:** `map_expr.get(key_expr, default=None)` returns the value for the given key, or the default (or NULL) if the key is not present [49]. For example, `t.properties.get("age", 0)` would attempt to get the `"age"` entry from the map, yielding 0 if missing [49]. - **Keys and Values:** `map_expr.keys()` returns an array of all keys in the map [50] [51], and `map_expr.values()` returns an array of all values [52] [48]. These arrays align by index (so element 0 of keys corresponds to element 0 of values). - **Length:** `map_expr.length()` gives the number of entries (size of the map) [42] [53]. - Some backends may support additional operations like checking if a key exists (`.contains(key)`, though often you'd use `.get` and check for null).

Usage example: If `t.metrics` is a `map<string, double>` containing various metric names to values, you can do: `t.metrics.get("conversion_rate")` to get the conversion_rate value for each row (result will be a double column, with null where the key is absent). You could unnest a map similarly to arrays:

`t.unnest("metrics")` would produce rows of key-value pairs (with columns like `key`, `value` for each map entry).

JSON and Semi-Structured Data

Ibis treats JSON data either as strings or as struct/array types depending on backend capabilities. Some backends (e.g. Postgres, MySQL) have JSON data types; Ibis may expose some functions to extract JSON fields. In general, if you have JSON text, you might parse it using backend-specific functions via `ibis.expr.operations`. Alternatively, if using DuckDB or others with struct support, you can cast JSON to a struct. The Ibis **JSON expressions** API (if available) would be listed under `ibis.expr.types.JSONValue` in docs, offering methods like `json_extract` or `json_get`. This is advanced and backend-specific – for AI agent usage, you might directly use raw SQL or UDFs to handle JSON.

Note: Complex types (struct/array/map) support varies by backend. For example, DuckDB, BigQuery, and Spark have good support for all; Postgres supports JSON/JSONB (which Ibis can map to struct), etc. If an operation on a complex type is not supported, Ibis will raise at compile time. Always consult the operation support matrix or docs for your backend if working heavily with nested types.

User-Defined Functions (UDFs and UD(A|W)Fs)

Ibis provides a powerful mechanism to define **User-Defined Functions (UDFs)** that can execute custom logic within the backend engine or in the client, bridging gaps where Ibis or SQL has no built-in operation. There are a few categories of UDFs in Ibis:

- **Built-in UDFs (SQL UDFs):** This is essentially a way to call **existing database functions** that Ibis doesn't have a high-level API for. By using the decorator `@ibis.udf.scalar.builtin`, you can wrap a backend-specific scalar function using Python syntax, and Ibis will compile it as a call to that function in SQL [54] [55]. For example, DuckDB has a function `hamming_distance(string, string)` that Ibis might not expose. You can do:

```python
@ibis.udf.scalar.builtin
def hamming(a: str, b: str) -> int:
    ...
```

The function body is not actually used – the `...` indicates we don't implement it in Python [56]. Ibis infers from the signature that it's a scalar operation taking two strings to an int. When you later use `hamming(t.col1, t.col2)` in an expression, Ibis will generate SQL like `HAMMING(col1, col2)` (using the function name, or a provided `name="..."` if you want to alias to a specific DB function name) [57] [58]. This provides an *escape hatch* to call any SQL function available in the backend that Ibis doesn't natively cover [59]. It's essentially syntactic sugar for writing a raw SQL function call, but with type checking and composability in Ibis.

- **Vectorized UDFs (pandas/UDF):** For backends that execute Python (e.g. the pandas, Dask, or Spark (pandas-on-Spark) backends), you can define UDFs that run using Python vectorized libraries. Use `@ibis.udf.scalar.pandas` to define a function that uses pandas or NumPy operations on columns. The function should accept pandas Series as arguments and return a Series or NumPy array of the result. For example, a pandas UDF to capitalize strings:

```
@ibis.udf.scalar.pandas
def str_cap(series: pd.Series[str]) -> pd.Series[str]:
    return series.str.capitalize()
```

Then `t.mutate(new_col = str_cap(t.some_string_col))` will under the hood apply the Python function on the pandas execution backend [60]. Similarly, `ibis.udf.scalar.pyarrow` lets you use PyArrow compute functions for vectorized UDFs (especially useful in PySpark or DataFusion backends). For instance, you might use `pyarrow.compute` library to implement a UDF, which can then run in those engines without Python loops [61] [62].

Warning: Ibis also allows `@ibis.udf.scalar.python` for non-vectorized Python UDFs [63] [64], but these execute row-by-row and can be **very slow** [65]. Use them only if absolutely necessary. Whenever possible, prefer pandas or pyarrow UDFs for vectorized performance [66].

- **Aggregate UDFs (UDAFs):** Ibis (as of v5) has *experimental* support for user-defined aggregations [67]. You can define a function that takes a Series (or array of values) and returns a scalar, decorated with `@ibis.udf.aggregate.pandas` or `@ibis.udf.aggregate.pyarrow` etc. For example, a UDAF that computes a custom statistical measure from a column can be registered. The API is similar to scalar UDFs but the function consumes the whole group of values. Once defined, you can use it in `agg()` just like built-in aggregates. Be aware these are experimental and may have limitations in certain backends.

- **Window UDFs (UDWFs):** Ibis does not currently expose a separate category for window UDFs; however, a combination of scalar UDF and using it over a window might achieve similar effect. Some backends (like Spark) have a concept of user-defined window functions, but in Ibis you would generally accomplish this by writing a scalar UDF and then calling it with `.over(window)` if it's something that can be applied per-row in a window. Truly stateful window UDFs are not directly in the Ibis API yet.

Using UDFs: After definition, a UDF can be called like a normal Ibis function in expressions [68] [69]. If it's a built-in UDF, Ibis will just generate the appropriate function call in SQL. If it's a pandas/pyarrow UDF, Ibis will execute it in the Python environment for backends that support that (e.g. the `pandas` backend or Spark with Arrow). Keep in mind: - **Registration:** For some backends like BigQuery, Ibis may register the UDF in the SQL engine (e.g., creating a temporary function) when executing the query. Others might inline the logic or execute on the client side. - **Type annotations:** It's important to annotate argument and return types on UDFs. Ibis uses these to infer the output expression's type [70]. For built-in UDFs, you *must* provide the return type annotation at least [71]. If an argument is a complex type (like an array of structs), you can use Ibis datatypes (from `ibis.expr.datatypes`) in the annotation [72]. - **Limitations:** Be cautious using UDFs in large-scale backends. A Python UDF in a SQL engine (like using the pandas backend for a part of the query) might require pulling data into Python. Pandas UDFs on large data could be memory-intensive. Always test performance. Some backends (Spark, Dask) can distribute the execution of pandas UDFs, but others (pandas, DuckDB) run them in-memory. - **Example – Builtin UDF:** DuckDB example from Ibis docs defines a UDF for DuckDB's `mismatches` function (which counts mismatched characters between strings) [54]. After definition, one could do:

```
expr = mismatches("duck", "luck")
con.execute(expr)  # returns 1
```

And you can see the generated SQL with `print(ibis.to_sql(expr, dialect="duckdb"))`, which would show: `SELECT MISMATCHES('duck', 'luck') AS "..."` [57].

- **Example – Pandas UDF:** Defining `@ibis.udf.scalar.pandas def add_one(x: pd.Series[int]) -> pd.Series[int]: return x + 1`. If you call `add_one(t.mycol)`, Ibis (when using pandas backend) will apply that function to the pandas Series for `mycol`. On a SQL backend, this UDF cannot run (unless Ibis knows how to translate it to SQL, which it generally doesn't), so use such UDFs only on backends that support Python execution.

Aggregate UDF example: (Experimental) You could do:

```
@ibis.udf.aggregate.pandas
def my_range(series: pd.Series[float]) -> float:
    return series.max() - series.min()

# usage:
result = table.group_by("group").agg(range_val=my_range(table.value))
```

If supported, this would compute the range of `value` within each group. (Currently, built-in aggregate UDFs exist primarily to wrap built-in DB functions similarly to scalar UDFs, and full Python aggregate UDF might only be supported on pandas-like backends.)

Backend Engines and Compilation

Ibis is **engine-agnostic**: you write expressions once, and you can execute them on many different backend engines. The supported backends (as of v5) include DuckDB (default), PostgreSQL/SQLite, MySQL, Spark, Pandas, Polars, BigQuery, Snowflake, Oracle, Dask, DataFusion, Trino, and more – nearly 20 in total [73]. Each backend has its own execution and SQL dialect, but Ibis abstracts that away.

Connecting to Backends

To use a backend, you typically call its `.connect()` method. For example: - DuckDB (default): `con = ibis.duckdb.connect("database.duckdb")`. If no path is given, it uses an in-memory DB. - PostgreSQL: `con = ibis.postgres.connect(database="dbname", host="localhost", user="me", password="pw")` (parameters vary per backend). - BigQuery: `con = ibis.bigquery.connect(project_id="myproj", dataset="mydataset")`. - Pandas: This backend has no external database; it just executes in-memory. You don't need to connect – operations default to the pandas backend if no engine is attached (and `ibis.options.default_backend` is set appropriately).

But you can explicitly do `con = ibis.pandas.connect()` if needed. - Polars: `con = ibis.polars.connect()` for using Polars engine in-memory [74]. - Spark (PySpark): `con = ibis.pyspark.connect(spark_session)` etc.

Multiple Backends: You can work with multiple connections simultaneously. Each table expression is bound to the backend that created it. For example, you could have `duck_con = ibis.duckdb.connect(); spark_con = ibis.pyspark.connect()`. If you call `duck_con.table("mytable")` you get a table on DuckDB; if you call `spark_con.table("mytable")`, that's a different expression on Spark (pointing to a Spark table). You can even join or union data across backends by pulling data to the client (though that defeats the purpose of pushdown). Typically, you keep expressions separate per backend or explicitly move data (see **Data In/Out** below).

Default Backend: Ibis has a concept of a default or global backend (`ibis.options.default_backend`). This is used by top-level functions like `ibis.read_csv` or `ibis.memtable` if you don't explicitly pass a connection. By default, Ibis v5+ sets DuckDB as the default engine [75]. This is why, for example, `ibis.read_parquet("file.parquet")` will use DuckDB under the hood to read that file (unless you changed the default backend).

You can change the default via `ibis.options.default_backend = spark_con` (assign an Ibis connection object). Then top-level operations will use that. Alternatively, many top-level functions accept a `backend=` argument to direct them.

Compiling and Executing Queries

Compilation to SQL: If you want to see the SQL that Ibis generates for an expression, use `ibis.to_sql(expr, dialect=<backend_dialect>)`. For example, `ibis.to_sql(result_expr, dialect="duckdb")` will print the SQL query string for the expression on DuckDB [57]. If the expression is already bound to a backend (e.g. it originated from `duck_con`), `ibis.to_sql(expr)` will often infer the dialect automatically, but it's safer to specify it. Note that some backends (pandas, Python) don't have SQL, so `to_sql` isn't applicable there.

For bound expressions, another approach is using the connection: many `con` objects have a `.compile(expr)` or `.to_sql(expr)` method. For example, `duck_con.compile(expr)` might return the SQL string. The exact API varies; the unified way is the global `ibis.to_sql`.

Executing expressions: To execute and retrieve results, use either the connection or the expression's `execute` method: - `con.execute(expr)` will execute the expression on that backend and return the result (as pandas DataFrame for tables or Python scalar for scalars) [76]. This is unified for SQL-based and pandas-based backends. - You can also do `expr.execute()` if the expression has a backend associated (e.g. after `t = con.table("X")`, `t.execute()` should fetch the table). However, in current Ibis it's recommended to use `con.execute(expr)` to be explicit about which backend to use. - For table expressions, you might prefer `con.to_pandas(expr)` which guarantees a pandas DataFrame output (especially if the backend might produce a different type, like PySpark DataFrame). For example, `df = con.to_pandas(table_expr)` [77]. Similarly, `con.to_pyarrow(expr)` would give a PyArrow Table if supported. - Some backends allow async execution or have additional methods (but those are advanced scenarios).

Expression Results: Executing a table expression yields a pandas DataFrame (for most backends) by default. Executing a scalar expression yields a Python scalar (or numpy scalar). Executing a column expression (like `t.col`) effectively fetches it as a pandas Series.

Partial Materialization (Head): Often you just want to see a few rows. Instead of calling `expr.execute()` on a large table, use `expr.head(n)` which is equivalent to adding a `limit(n)` and then executing. In interactive mode, printing `expr.head(n)` will automatically fetch only `n` rows (because of the default limit), which is efficient.

Verbose Logging: Ibis can log the SQL and other activity for debugging. Set `ibis.options.verbose = True` to print every query being executed [78]. By default, it prints to stdout; you can redirect log messages by assigning a function to `ibis.options.verbose_log` [79]. This is useful to trace what Ibis is doing, especially in complex workflows. In Jupyter, you might see the SQL printed if verbose is on whenever an execution happens.

Backend-Specific Notes and Compilation Differences

Different engines have different SQL dialects and capabilities. Ibis tries to cover common SQL, but some differences to note: - **Temporal functions:** Some backends might not support a specific date/time function. Ibis uses a normalization layer (often via `sqlglot`) to translate expressions, but e.g. truncating timestamps or extracting fields could differ. Check the operation support if something fails. - **Analytics:** Window functions are widely available in analytical databases (DuckDB, Postgres, etc.) but not all backends (e.g., some NoSQL or older ones) support them. Ibis will raise if you attempt an unsupported operation. - **Data types:** Backends vary in type systems (e.g., MySQL might not have a boolean type, DuckDB has UUID, etc). Ibis maps types internally. In edge cases you might need to cast explicitly using `expr.cast(type)`. - **Backend hints:** If you need to use a backend-specific SQL snippet, you can use `ibis.raw_sql("...")` or in some cases `con.sql("SELECT ...")` to execute a custom SQL and get an Ibis table (for engines that support it). There's also `ibis.expr.operations.SQLStringView` for creating a view from a SQL string [80].

Ibis maintains an **operation support matrix** tracking which features are implemented for each backend [81]. Due to SQL dialect differences and varying backend implementations, not every Ibis function works on every backend. For example, some backends might lack geospatial operations or advanced analytics. If you use a feature not supported on a backend, you'll get an error at compile time (an `UnsupportedOperationError` typically). Consult the [Ibis operation support matrix][**] (on Ibis docs site) to see coverage. In general, mature backends like DuckDB, Postgres, and BigQuery have high coverage, whereas newer or niche backends might miss some operations. This is a good area to contribute if you need something – implement the operation for that backend.

Schema and Type Introspection

Schemas: Each table expression has an associated schema (`ibis.Schema`) describing its column names and types. You can get it via `table.schema()` [82]. For example:

```
print(t.schema())
```

might output:

```
ibis.Schema {
  species           string
  island            string
  bill_length_mm    float64
  ...
}
```

[82]. This is similar to a SQL DESCRIBE table. The schema object has attributes like `names` (list of column names) and `types` (list of datatypes). You can also access `table.columns` (list of names) or iterate `table.schema().items()` for name/type pairs.

Datatypes: Ibis has its own type system (`ibis.expr.datatypes`). The `ibis.schema({...})` function can create a schema from a dict of names to types (types as strings or `ibis.dtype` objects). Example: `schema = ibis.schema({"a": "int64", "b": "array<string>"})`. You can use this to create unbound tables or to assert a schema when reading data.

If you need to programmatically inspect a column's type, you can use `expr.type()` which returns an Ibis `DataType` instance. You can compare it to known types, e.g. `expr.type().is_integer()`.

Examples of type usage:

```
for name, dtype in t.schema().items():
    if dtype == dt.int64:
        # do something for int columns
```

The `ibis.expr.datatypes` module is usually imported as `import ibis.expr.datatypes as dt`. It has factory functions like `dt.Array(dt.string)` or simply `ibis.dtype("array<string>")`.

Printing info: In interactive mode, printing a table shows a preview of data plus the schema (the header with types). You can also do `table.info()` which returns a table expression of summarized info (similar to pandas .info, but Ibis v5 changed this to return an expression). Concretely, `t.info()` yields a small table with columns like name, type, #nulls, etc., which you can execute to see stats. (Note: After v5, `info()` became experimental and might require an execution to get meaningful info. Many prefer just `print(t.schema())`.)

Input/Output: Reading and Writing Data

Ibis can directly read from and write to various data sources in addition to database tables. This makes it a convenient single interface for files and DBs.

In-memory Data (Pandas/Arrow)

- **Memtables:** Use `ibis.memtable(data)` to create a table expression from in-memory data (Pandas DataFrame, Python list of dicts, Arrow table, etc.). This is great for quick analysis or joining small data with large remote tables. Example: `df = pandas.DataFrame(...); t = ibis.memtable(df)` [83]. The resulting `t` is a table expression (default backend will likely be DuckDB or pandas). If using DuckDB, Ibis will load that DataFrame into DuckDB internally (often via Arrow). Memtables are executed by materializing the data in the execution engine. Under the hood, for SQL backends Ibis often creates a temporary table (DuckDB, Snowflake, etc. support this).
- **Conversion to Pandas:** `table_expr.to_pandas()` will execute and return a pandas DataFrame [84]. This is equivalent to `con.execute(table_expr)` but explicitly requests a pandas object (which for many backends is the default). If the backend returns Arrow, Ibis will convert to pandas if needed. There's also `to_pyarrow()` to get an Arrow Table instead [85].
- **DataFrame Protocols:** Ibis table expressions implement the `__dataframe__` and `__array__` protocols [86], meaning you can often directly pass an Ibis table to libraries expecting a DataFrame (they will under-the-hood call `to_pandas()` or similar). For example, `pd.DataFrame(t)` might try to use `__dataframe__` to convert to pandas. Similarly `numpy.array(t)` for a 1D result might trigger `__array__`. This interoperability is being developed across libraries.

Reading Files

Ibis can read CSV, Parquet, and other files directly, if the default or specified backend supports it. For example: - `ibis.read_csv(path)` – returns a table expression reading a CSV file [87]. By default it uses DuckDB (if default backend is DuckDB or a file-supporting engine). DuckDB will infer types from the CSV. You can pass backend= or engine options if needed. - `ibis.read_parquet(path)` – returns a table expr for a Parquet file [88]. Parquet has schema information, so it's straightforward. - `ibis.read_delta(path)` – for Delta Lake tables (if using PySpark or DuckDB with deltalake extension) [89]. - Some backends support `read_json` for NDJSON files, or `read_avro` etc., if implemented.

These top-level functions often proxy to the backend's implementation. For instance, DuckDB's backend can create an external table for a CSV or Parquet. If reading a large file, you can treat the returned expression like any other – apply filters, etc., and only the needed data will be read (predicate pushdown happens in some cases).

Writing Files

You can write the result of an expression to a file via methods on the table expression: - `expr.to_csv('output.csv')` – executes the expression and materializes it as a CSV file [90]. This requires the backend to support writing CSV (DuckDB does, using its COPY TO). By default, DuckDB will write a single CSV file. Be mindful if your data is large. - `expr.to_parquet('output.parquet')` – writes to Parquet [88]. Parquet writing is supported in DuckDB, Polars, etc. Parquet is often preferred for large data. - `expr.to_delta('path.delta', mode="overwrite")` – if using a Delta Lake capable backend, writes to a Delta table [89]. - These functions may have backend-specific options, e.g. partitioning, compression – check Ibis docs or pass arguments if supported. For example, DuckDB allows `partition_by=["col"]` in `to_parquet`.

Keep in mind, writing large datasets will pull data through the backend. E.g., if using DuckDB, it can write Parquet directly without pulling data into Python. If using the pandas backend, writing Parquet will use `pyarrow` or `pandas` to do so (and likely require all data in memory). Always prefer pushing the writing to a capable backend like DuckDB.

Database Tables and Other Sources

If you are connected to a SQL backend (e.g., Postgres, MySQL, BigQuery), you can access existing tables via `con.table("name")`. This returns an Ibis table expression for that remote table. You can also list available tables (e.g. `con.list_tables()` or using the catalog in some backends). Once you have an expression, all the Ibis operations apply and will be compiled to SQL on that database.

Creating tables: On some backends, you can create new tables from Ibis expressions: - `con.create_table("newname", schema=..., overwrite=True)` – creates an empty table with the given schema on the backend [91]. - `con.insert("tablename", expr)` – inserts the results of an expression into a table (or creates and inserts) [92]. For example, after computing `expr = ...` you can do `con.create_table("myresult", schema=expr.schema()); con.insert("myresult", expr)` [91] [92]. In some cases `create_table(..., expr)` is also available, which creates a new table from the results of an expression in one step (depending on backend). - For file-based engines (DuckDB, etc.), writing to Parquet/CSV as above is often easier than `insert`.

Importing/Exporting between Backends

Because Ibis can work with multiple backends, you can **transfer data** using Ibis as well: - You can fetch data from one backend into Python (as pandas) and then use `ibis.memtable` or `con2.insert` to send it to another backend. For moderate data sizes this is fine. - Some backends can read from others directly (e.g. DuckDB can query remote Postgres or read Parquet from cloud storage). Ibis doesn't fully automate cross-backend queries yet, but you can use raw connections or `ibis.native` to leverage that if needed. - The design of Ibis encourages pushing computation to the data rather than moving data. So ideally, connect to the system where data resides and perform expression there. If you must combine data from two sources, consider pulling both into a common engine (like DuckDB or pandas) via Ibis and then joining/combining in that engine.

Configuration and Options

Ibis has a global `ibis.options` object for configuration settings [93]. Some of the important options:

- **Interactive:** `ibis.options.interactive` (default `False`). As discussed, if True, simply evaluating an expression triggers execution [4]. This is convenient for notebooks but should be False in scripts to avoid accidental execution.
- **Default Backend:** `ibis.options.default_backend`. Controls which backend is used for top-level operations (when an expression isn't explicitly tied to a connection) [75]. By default, an instance of DuckDB is set as the default. You can assign a connection here or use `ibis.set_backend(con)`.
- **SQL Options:** `ibis.options.sql.default_limit` (default 10,000) limits the number of rows fetched on executing a table without an explicit limit [17]. Set to `None` to disable (but be cautious: a

large query could pull millions of rows). Another SQL option is `ibis.options.sql.default_dialect` (default `"duckdb"`) which sets a default dialect for printing SQL when it can't be inferred [94]. Also, `ibis.options.sql.fuse_selects` (boolean, default True) controls whether Ibis tries to fuse multiple nested select statements into one for cleaner SQL [94].

- **Repr Options:** `ibis.options.repr.table_rows` and `repr.table_columns` control how many rows and columns are shown when pretty-printing a table in interactive mode [95] [96]. By default it shows up to 10 rows and all columns that fit. You can adjust these for a fuller preview or more compact output. There's also `repr.show_types` (bool) to show data types in the column headers when printing [97] (in the ASCII table output).
- **Verbose/Logging:** As mentioned, `ibis.options.verbose` and `verbose_log` allow seeing the SQL queries and other debug info [78].
- **Graphviz debugging:** Ibis can visualize the query plan graph. If you call `expr.visualize()` it will produce a graph (requires graphviz). This is more of a development feature. Under the hood, it may use `ibis.options.graphviz` settings.
- **Precision and Pandas interop:** If using pandas backend, options like `ibis.options.pandas.tz` or others could exist (though not commonly needed by end users).
- **UDF config:** Not much global config, but note if using Spark, you might need to ensure Arrow is enabled for pandas UDFs. Ibis tries to handle config like setting `SparkSession.conf` for Arrow when needed.

In summary, the default options are sensible for most uses. Typically, one might tweak `interactive` (on in notebooks, off in scripts) and `sql.default_limit` if the 10k row cap is an issue. Also, set `verbose=True` when debugging SQL generation or performance issues to see what queries are being run.

Examples of Usage Across Backends

One strength of Ibis is the ability to write a query once and execute on different engines. For instance, consider an expression defined on an unbound table or created via Ibis high-level API:

```
# Define transformation on an example dataset (Palmer Penguins)
t = ibis.table(
  schema=dict(species="string", island="string", bill_length_mm="float64",
              bill_depth_mm="float64", flipper_length_mm="int64",
body_mass_g="int64",
              sex="string", year="int64"),
  name="penguins"
)
expr = (t.group_by(["species", "island"])
          .agg(count = t.species.count())
```

```
        .order_by(_.count.desc())
        .limit(3))
```

This `expr` is a deferred query to get the top 3 species+island combinations by count. We can execute it on any backend that has the `penguins` data:

- **DuckDB:**

  ```
  con = ibis.duckdb.connect()
  con.register("penguins", pandas_df)           # register pandas dataframe
  as a DuckDB table
  result = con.execute(expr)                    # executes the query on
  DuckDB
  ```

  or simply `ibis.options.default_backend = con; expr.execute()`. DuckDB will run the group by and produce the result.

- **BigQuery:**

  ```
  con = ibis.bigquery.connect(project_id="myproj", dataset="samples")
  con.create_table("penguins", schema=expr.schema(), data=pandas_df)  #
  upload data (if not already in BQ)
  result = con.execute(expr)
  ```

  Ibis will generate BigQuery SQL for the aggregation and fetch the result.

- **Pandas:**

  ```
  con = ibis.pandas.connect({"penguins": pandas_df})
  df = con.execute(expr)
  ```

  This will use pandas under the hood (the group_by will become a pandas groupby operation).

In all cases, `expr` itself did not change – only the execution target changed. This demonstrates Ibis' portability: the same analytical logic can run in-process via DuckDB or out-of-process on a distributed SQL engine, depending on data size and context [98] [73]. The only requirement is that the backend supports the operations used in the expression (in this case, group_by, count, order_by, limit – which all backends do). If we had used a more exotic feature (say a geospatial function), not all backends would work.

Ibis ensures that if an operation is unsupported, you find out at compile time (when calling execute or to_sql). This allows writing engine-agnostic code and then handling exceptions for unsupported features as needed (possibly with backend-specific code paths as fallbacks).

Finally, when integrating Ibis in code, remember that it is **not a DataFrame itself** but a query builder. It excels at pushing computation to databases and query engines. Use Ibis expressions for heavy data lifting, and bring back small results to Python for final tasks (or further use in libraries like pandas, numpy, etc.). This approach leverages the power of AI-friendly engines (like DuckDB's vectorized execution, or BigQuery's distributed computing) while allowing you to write in Python with type checking and high-level abstractions.

References: This reference draws from the Ibis documentation and examples, including usage of deferred expressions [9] [10], table operations like joins [18] [19], and complex data type handling [40], as well as Ibis configuration options [78] and multi-backend execution best practices [73]. For a comprehensive list of supported operations per backend, see the official Ibis operation support matrix [81].

---

[1] [2] [3] [4] [17] [75] [78] [79] [93] basics – Ibis
https://ibis-project.org/how-to/configure/basics

[5] [6] [7] [73] [77] [91] [92] [98] Write and execute unbound expressions – Ibis
https://ibis-project.org/how-to/extending/unbound_expression

[8] [9] [10] chain_expressions – Ibis
https://ibis-project.org/how-to/analytics/chain_expressions

[11] [12] [14] [15] [16] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [82] expression-tables – Ibis
https://ibis-project.org/reference/expression-tables

[13] [38] [39] ffill_bfill_w_window – Ibis
https://ibis-project.org/how-to/timeseries/ffill_bfill_w_window.html

[36] python - Use Ibis to filter table to row with largest value in each group
https://stackoverflow.com/questions/78722158/use-ibis-to-filter-table-to-row-with-largest-value-in-each-group

[37] Use ibis-framework to compute shifts (lags) in dataframe
https://stackoverflow.com/questions/78631285/use-ibis-framework-to-compute-shifts-lags-in-dataframe

[40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] expression-collections – Ibis
https://ibis-project.org/reference/expression-collections

[54] [55] [56] [57] [58] [59] [68] [69] [76] builtin – Ibis
https://ibis-project.org/how-to/extending/builtin

[60] [61] [62] [63] [64] [65] [66] [67] [70] [71] [72] scalar-udfs – Ibis
https://ibis-project.org/reference/scalar-udfs

[74] multiple-backends – Ibis
https://ibis-project.org/how-to/input-output/multiple-backends

[80] operations – Ibis
https://ibis-project.org/reference/operations

[81] Operation support matrix – Ibis
https://ibis-project.org/backends/support/matrix

[83] [84] [85] [86] [87] [88] [89] [90] basics – Ibis
https://ibis-project.org/how-to/input-output/basics

[94] sql – Ibis
https://ibis-project.org/reference/sql

[95] [96] [97] repr – Ibis
https://ibis-project.org/reference/repr